
Advances in Database Technology — EDBT 2018

21st International Conference
on Extending Database Technology
Vienna, Austria, March 26–29, 2018
Proceedings

Editors

Michael Böhlen
Reinhard Pichler
Norman May
Erhard Rahm
Shan-Hung Wu
Katja Hose



Advances in Database Technology – EDBT 2018
Proceedings of the 21st International Conference
on Extending Database Technology
Vienna, Austria, March 26–29, 2018

Series ISSN: 2367-2005

Editors

Michael Böhlen, University of Zürich, Switzerland
Reinhard Pichler, TU Wien, Austria
Norman May, SAP Research, Germany
Erhard Rahm, University of Leipzig, Germany
Shan-Hung Wu, National Tsing Hua University, Taiwan
Katja Hose, Aalborg University, Denmark



OpenProceedings.org
University of Konstanz
University Library
78457 Konstanz, Germany

COPYRIGHT NOTICE: Copyright © 2018 by the authors of the individual papers.

Distribution of all material contained in this volume is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0

OpenProceedings ISBN: 978-3-89318-078-3

DOI of this front matter: 10.5441/002/edbt.2018.01

Foreword

The International Conference on Extending Database Technology (EDBT) is a leading international forum for database researchers, developers, and users to present and discuss cutting-edge ideas, and to exchange techniques, tools, and experiences related to data management. Data management is an essential enabling technology for scientific, business, and social communities. It is driven by the requirements of applications across many scientific, business and social communities, and runs on diverse technical platforms associated with the web, enterprises, clouds and mobile devices. The database community has a continuing tradition of contributing with models, algorithms and architectures to the set of tools and applications that enable day-to-day functioning of our societies. Faced with the broad challenges of today's applications, data management technology constantly broadens its reach, exploiting new hardware and software to achieve innovative results.

EDBT 2018 solicited submissions of original research contributions, descriptions of industrial solutions and applications, and proposals for tutorials and software demonstrations. We encouraged submissions of research papers related to all aspects of data management. In addition to regular research paper submissions, EDBT 2018 again encouraged the submission of short research papers, which includes visionary papers that provide a forum for the identification and discussion of new or emerging areas, innovative or risky approaches, or emerging applications that require extensions of established techniques. Short papers are presented as posters at plenary poster sessions of the conference. This provides an excellent opportunity to describe significant work in progress or research that is best communicated interactively and fosters discussions.

The program committees of EDBT accepted 35 out of 142 submitted regular research papers, resulting in an acceptance rate of 24.6% for the research track; 23 out of 84 submitted short research papers, resulting in an acceptance rate of 27.4% for short research papers; 16 out of 37 demos, resulting in an acceptance rate of 43.2% for the demonstration track; and 10 out of 28 industrial and application papers, resulting in an acceptance rate of 35.7% for industrial and application papers. The papers will be presented in twelve research paper sessions, three industrial and application sessions, as well as two plenary poster and demonstration sessions. The program additionally features four workshops, one of which is the well-established DOLAP workshop that has successfully been co-located with EDBT since many years. Finally, the conference program includes four tutorials and an EDBT and ICDT joint session on research challenges.

I would like to thank all authors for their contributions: a successful conference crucially depends on high-quality submissions. I also would like to thank all reviewers for serving on the EDBT 2018 program committee, in particular for the high quality and timely handling of all reviews and discussions. This community service requires a lot of work on a tight schedule, and is what makes our research community function and ensures the sustained impact of our research. Thanks to this effort we can look forward to an exciting program and attractive EDBT conference in Vienna from March 26-29, 2018. A warm thanks to Norman Paton and Divesh Srivastava for serving on the Test-of-Time Award committee to select the paper from EDBT 2008 that has had the most lasting influence. The committee selected the paper Social ties and their relevance to churn in mobile telecom networks by Koustuv Dasgupta, Rahul Singh, Balaji Viswanathan, Dipanjan Chakraborty, Sougata Mukherjea, Amit A. Nanavati, and Anupam Josi for the test-of-time award.

Lei Chen, Wolfgang Lehner and Kian-Lee Tan generously accepted to serve on the Best Paper committee. As best paper, the committee selected the paper Temporally-Biased Sampling for Online Model Management by Brian Hentschel from Harvard University, Peter Haas from the University of Massachusetts Amherst, and Yuanyuan Tian from IBM Almaden Research Center. The EDBT best paper runner-up was awarded to the paper GeoAlign: Interpolating Aggregates over Unaligned Partitions by Jie Song from the University of Michigan, Danai Koutra from the University of Michigan, Murali Mani from the University of Michigan, Flint, and H. Jagadish from the University of Michigan. Congratulations to the awardees and a warm thanks to the committee members for their work.

The EDBT 2018 program is the result of the joint effort of many people who shared their experience and time to contribute to the EDBT 2018 program and make the conference a success. Norman May served as PC chair for industrial and application papers; Shan-Hung as PC chair for the demonstration track; Erhard Rahm as tutorial chair; Nikolaus Augsten as workshop chair; and Dan Olteanu as challenge session organizer. My warmest thank to all these people.

The general chair, Reinhard Pichler and the local organizers worked hard to make all necessary arrangements for a successful event. Special thanks to Katja Hose, the proceedings chair; Dimitris Sacharidis, the sponsorship chair; and Shqiponja Ahmetaj, Markus Kröll, and Wolfgang Fischl, the website chairs, for tirelessly finding solutions for all our needs and making things happen. Norman Paton was most helpful in advising and coordinating with the EDBT Executive Board.

I hope that you find EDBT 2018 inspiring, informative, and enjoyable and look forward to meeting you in Vienna.

Michael Böhlen
EDBT 2018 Program Chair

Program Committee Members

Research Program Committee

Ahmed Eldawy, UC Riverside
Angela Bonifati, U Lyon
Anton Dignös, Free U Bozen-Bolzano
Arash Termehchy, Oregon State U
Arijit Khan, Nanyang Techn. U
Arnab Bhattacharya, IIT Kanpur
Bernhard Seeger, U Marburg
Bin Cui, Peking U
Boris Glavic, Illinois Inst. of Techn.
Carsten Binning, Brown U
Ce Zhang, ETH Zurich
Curtis Dyreson, Utah State U
Daniele Dell’Aglia, U Zürich
Davide Martinenghi, Politecnico di Milano
Denilson Barbosa, U Alberta
Elena Ferrari, U Insubria
Floris Geerts, U Antwerp
Giansalvatore Mecca, U Basilicata
Gottfried Vossen, U Münster
Guoliang Li, Tsinghua U
Heiko Schuldt, U Basel
Hong Va Leong, Hong Kong Polytechnic U
Jens Teubner, TU Dortmund
K. Selçuk Candan, Arizona State U
Karl Aberer, EPF Lausanne
Kian-Lee Tan, National U Singapore
Kjetil Norvag, NTNU
Klaus Berberich, MPI Informatics
Klemens Böhm, Karlsruhe Inst. Technology
Kostas Stefanidis, U Tampere
Kyriakos Mouratidis, Singapore Mgmt. U
Lei Chen, Hong Kong U Sc. & Tech.
Man Lung Yiu, Hong Kong Polytechn. U
Marc Spaniol, U Caen Basse-Normandie
Martin Theobald, U Luxembourg
Matthias Renz, George Mason U
Maurice van Keulen, UTwente
Mohamed Eltabakh, Worcester PI
Mohammad Sadoghi, Purdue U
Mourad Ouzzani, Qatar Cmptg. Res. Inst.
Nikos Pelekis, U Piraeus
Nikos Mamoulis, U Ioannina
Panagiotis Papapetrou, Stockholm U
Panagiotis Bouros, Aarhus U
Panagiotis Karras, Aalborg U
Paolo Papotti, EURECOM
Periklis Andritsos, U Toronto
Peter Triantafillou, U Glasgow
Philippe Cudre-Mauroux, U Fribourg
Pierre Senellart, ENS
Rainer Gemulla, U Mannheim
Ralf Hartmut Güting, FU Hagen
Reynold Cheng, U Hong Kong
Riccardo Torlone, U Roma Tre
Ryan Stutsman, U Utah
Sabrina De Capitani di Vimercati, U Milano
Sebastian Michel, TU Kaiserslautern
Senjuti Roy, New Jersey Inst. Techn.
Sherif Sakr, U New South Wales
Sourav S Bhowmick, Nanyang Techn. U
Stefan Manegold, CWI Amsterdam
Stratis Viglas, U Edinburgh
Sven Helmer, Free U Bozen-Bolzano
Themis Palpanas, Paris Descartes U
Theodore Johnson, AT&T Labs
Thomas Seidl, LMU München
Timos Sellis, Swinburne UT
Tore Risch, Uppsalla U
Torsten Grust, U Tübingen
Ulf Leser, HU Berlin
Verena Kantere, U Geneva
Wai Kit Wong, Hang Seng Mgmt. College
Walid Aref, Purdue U
Wei-Shinn Ku, Auburn U
Wolfgang Lehner, TU Dresden
Wook-Shin Han, Postech
Xin Huang, Hong Kong Baptist U
Yang Cao, U Edinburgh
Yannis Theodoridis, U Piraeus
Yannis Velegarakis, U Trento
Yaron Kanza, Technion
Ying Yang, State U New York
Ying Zhang, UT Sydney

Industrial Program Committee

Berthold Reinwald, IBM Research
Carl-Christian Kanne, Workday
Christian Mathis, SAP
Danica Porobic, Oracle
Eric Simon, SAP-BO
Florian Funke, Snowflake
Jörg Schad, Mesosphere
Manuel Then, Tableau
Martin Grund, Amazon
Matthias Brantner, Oracle
Matthias Boehm, IBM
Philipp Unterbrunner, Facebook
Pinar Tozun, IBM
Stefan Mandl, Exasol
Tobias Muehlbauer, Muehlbauer Tableau

Demonstration Program Committee

Alessandro Campi, Politecnico di Milano
Alfredo Cuzzocrea, U Trieste & ICAR-CNR
Anisoara Nica, SAP SE Waterloo
Berthold Reinwald, IBM Research
Danai Symeonidou, French Nat.Inst. Agricultural Res.
Demetris Zeinalipour, MPI Informatics & U Cyprus
Dirk Habich, TU Dresden
Elisa Quintarelli, Politecnico di Milano
Eric Lo, Chinese U Hong Kong
Ernest Teniente, UP Catalunya
George Fletcher, TU Eindhoven
Guoliang Li, Tsinghua U
Haruo Yokota, Tokyo Inst. Technology
Kai-Uwe Sattler, TU Ilmenau
Katja Hose, Aalborg U
Leopoldo Bertossi, U Carleton
Letizia Tanca, Politecnico di Milano
Michael Gertz, Heidelberg U
Neil Conway, Mesosphere
Norman Paton, U Manchester
Tony Tan, Nat. Taiwan U
Vasilis Vassalos, Athens U Economics & Business
Yi-Leh Wu, Nat. Taiwan U Sci. & Techn.
Yingyi Bu, Couchbase
Zhifeng Bao, RMIT U

External Reviewers

Abdallah Arioua, Lyon 1 U
Alessio Conte, Nat.Inst. Informatics Tokyo
Amit Gupta, EPF Lausanne
Andreas Spitz, Heidelberg U
Anna Beer, LMU Munich
Ben McCamish, Oregon State U
Bo Tang, Southern U Sc. & Techn. China
Chenhao Ma, U Hong Kong
Christian Beilschmidt, U Marburg
Danhao Ding, U Hong Kong
Daniyal Kazempour, LMU Munich
Denis Martins, U Münster
Dimitris Sacharidis, TU Wien
Donatella Firmani, U Roma Tre

Donatello Santoro, U Basilicata
Duong Chi Thang, EPF Lausanne
Evgeniy Faerman, LMU Munich
Fabio Valdés, FU Hagen
Fan Zhang, U New South Wales
Florian Funke, Snowflake Computing
Florian Richter, LMU Munich
Giulia Pretti, U Trento
Hamza Harkous, EPF Lausanne
Hanchen Wang, UT Sydney
Harris Georgiou, U Piraeus
Jan-Kristof Nidzwetzki, FU Hagen
Janina Sontheim, LMU Munich
Jens Lechtenbörger, U Münster
Jiafeng Hu, U Hong Kong
Jilian Zhang, Jinan U
Johannes Droenner, U Marburg
Jose Picado, Oregon State U
Julian Busch, LMU Munich
Kewen Liao, Swinburne UT
Kiril Panev, TU Kaiserslautern
Koninika Pal, TU Kaiserslautern
Kostas Patroumpas, IMIS Athena Res. Ctr.
Kostas Zoumpatianos, Harvard U
Leschek Homann, U Münster
Loc Do, U Hong Kong
Manuel Hoffmann, TU Kaiserslautern
Matteo Lissandrini, U Trento
Max Berrendorf, LMU Munich
Michael Körber, U Marburg
Michael Mattig, U Marburg
Michele Linardi, Paris Descartes U
Mingjie Li, UT Sydney
Nicolas Pflanzl, U Münster
Nikolaus Glombiewski, U Marburg
Panayiotis Smeros, EPF Lausanne
Parisa Ataie, Oregon State U
Patrick Schaefer, HU Berlin
Pavlos Paraskevopoulos, George Mason U
Peipei Yi, Hong Kong Baptist U
Prithvi Sen, IBM Almaden
Ruey-Cheng Chen, RMIT U
Spencer Pearson, Purdue U
Suyash Gupta, UC Davis
Tam Nguyen Thanh, EPF Lausanne
Thamir Qadah, Purdue U
Theodoros Chondrogiannis, U Konstanz
Thomas Fober, U Marburg
Thomas Krause, HU Berlin
Ugo Comignani, Lyon 1 U
Vahid Ghadakchi, Oregon State U
Yifeng Lu, LMU Munich
Yixiang Fang, U Hong Kong
Yodsawalai Chodpathumwan, Oregon State U
Yongrui Qin, U Huddersfield
You Peng, U New South Wales
Yuli Jiang, Chinese U Hong Kong
Zhipeng Huang, U Hong Kong
Zhipeng Zhang, Peking U
Zichen Zhu, U Hong Kong

Test-of-Time Award

Since 2014, the Extended Database Technology (EDBT) Conference awards the EDBT test-of-time award, with the goal of recognizing papers presented at EDBT Conferences that have had the most impact in terms of research, methodology, conceptual contribution, or transfer to practice.

This year the award has been given to a paper from the EDBT 2008 Conference in Nantes, France. The award was bestowed upon the paper:

Social ties and their relevance to churn in mobile telecom networks

by Koustuv Dasgupta, Rahul Singh, Balaji Viswanathan, Dipanjan Chakraborty, Sougata Mukherjea, Amit A. Nanavati and Anupam Josi

published in the EDBT 2008 Proceedings, pp. 668–677, DOI: 10.1145/1353343.1353424.

This industry track paper reports on an analysis of mobile telecoms data, with a view to predicting which customers will leave. The analysis involves commercial mobile telephony data, in which nodes are customers and edges represent calls. The hypothesis tested is that it is possible to predict who will leave a network based on earlier departures among their connections. The main technique investigated is the use of spreading activation, to predict the heat of nodes based on the heat of connected nodes. It is shown how the approach based on connections is more effective than classification techniques based on other properties of the nodes. As a result, the paper provides early and compelling experience on the combination an important real problem (churn in mobile telecom networks) with a powerful technique (social ties) and applies it on large real data (telecom operator network over 4 months). The approach has influenced many subsequent studies, for the same problem, but also for analyses involving different types of network and different hypotheses. Social network analysis continues as an important and active area ten years later, and this paper continues to be widely cited.

The EDBT 2018 Test-of-Time Award Committee consisted of Michael Böhlen, Divesh Srivastava and Norman Paton. The EDBT Test-of-Time award for 2018 will be presented during the EDBT/ICDT 2018 Conference, March 26–29, in Vienna, Austria (<http://edbticdt2018.at>).

Best Paper Award

The best paper award was bestowed upon the paper:

Temporally-Biased Sampling for Online Model Management

by Brian Hentschel from Harvard University, Peter Haas from the University of Massachusetts Amherst, and Yuanyuan Tian from the IBM Almaden Research Center. DOI: 10.5441/002/edbt.2018.11

The paper proposes a temporally-biased sampling method for a stream of batches that weighs recent data items more heavily. The inclusion probabilities of data items decay exponentially over time. The authors introduce a reservoir-based temporally-biased sampling method that asserts an upper bound on the sample size while keeping the decay of the sample predictable. The problem is well motivated and described, and the paper offers an excellent solution that is formalized precisely, is robust in the presence of evolving data, and has been implemented and evaluated for a distributed setting.

The best paper runner-up award was bestowed upon the paper:

GeoAlign: Interpolating Aggregates over Unaligned Partitions

by Jie Song from the University of Michigan, Danai Koutra from the University of Michigan, Murali Mani from the University of Michigan, Flint, and H. Jagadish from the University of Michigan. DOI: 10.5441/002/edbt.2018.32

This paper introduces a novel technique to integrate geographical summaries over unaligned geographical regions, e.g., counties and ZIP codes. While traditional techniques assume that the data in each region is uniformly distributed, the proposed approach infers the distribution based on other datasets. The proposed idea is novel, refreshing, and nicely motivated. The described solutions are practical, have been implemented and evaluated, and there is good potential for follow-up work.

The EDBT 2018 Best Paper Award Committee consisted of Michael Böhlen, Lei Chen, Wolfgang Lehner, and Kian-Lee Tan. The EDBT Best Paper Awards for 2018 will be presented during the EDBT/ICDT 2018 Conference, March 26-29, in Vienna, Austria (<http://edbticdt2018.at>).

Table of Contents

Foreword	i
Program Committee Members	ii
Test-of-Time Award	iv
Best Paper Award	v
Table of Contents	vi
Research Papers	
ID Repair for Trajectories with Transition Graphs <i>Xingcan Cui, Xiaohui Yu, Xiaofang Zhou, Jiong Guo</i>	1
MTBase: Optimizing Cross-Tenant Database Queries <i>Lucas Braun, Renato Marroquin, Ken Tsay, Donald Kossmann</i>	13
Extending In-Memory Relational Database Engines with Native Graph Support <i>Mohamed Hassan, Tatiana Kuznetsova, Hyun Chai Jeong, Walid Aref, Mohammad Sadoghi</i>	25
Sequenced Route Query with Semantic Hierarchy <i>Yuya Sasaki, Yoshiharu Ishikawa, Yasuhiro Fujiwara, Makoto Onizuka</i>	37
On Complexity and Efficiency of Mutual Information Estimation on Static and Dynamic Data <i>Michael Vollmer, Ignaz Rutter, Klemens Böhm</i>	49
Finding All Maximal Connected s-Cliques in Social Networks <i>Rachel Behar, Sara Cohen</i>	61
Summarization Algorithms for Record Linkage <i>Dimitrios Karapiperis, Aris Gkoulalas-Divanis, Vassilios S. Verykios</i>	73
Continuous Monitoring of Pareto Frontiers on Partially Ordered Attributes for Many Users <i>Afroza Sultana, Chengkai Li</i>	85
Optimizing Selection Processing for Encrypted Database using Past Result Knowledge Base <i>Wai Kit Wong, Kwok Wai Wong, Ho-Yin Yue</i>	97
Temporally-Biased Sampling for Online Model Management <i>Brian Hentschel, Peter J. Haas, Yuanyuan Tian</i>	109
Detecting Database File Tampering through Page Carving <i>James Wagner, Alexander Rasin, Tanu Malik, Karen Heart, Jacob Furst, Jonathan Grier</i>	121
User-guided Repairing of Inconsistent Knowledge Bases <i>Abdallah Arioua, Angela Bonifati</i>	133
Synchronous Multi-GPU Training for Deep Learning with Low-Precision Communications: An Empirical Study <i>Demjan Grubic, Leo Tam, Dan Alistarh, Ce Zhang</i>	145
EasyCommit: A Non-blocking Two-phase Commit Protocol <i>Suyash Gupta, Mohammad Sadoghi</i>	157
Beyond Frequencies: Graph Pattern Mining in Multi-weighted Graphs <i>Giulia Preti, Matteo Lissandrini, Davide Mottin, Yannis Velegrakis</i>	169
Scalable Evaluation of k-NN Queries on Large Uncertain Graphs <i>Xiaodong Li, Reynold Cheng, Yixiang Fang, Jiafeng Hu, Silviu Maniu</i>	181

MatchCatcher: A Debugger for Blocking in Entity Matching <i>Han Li, Pradap Konda, Paul Suganthan G C, Anhai Doan, Benjamin Snyder, Youngchoon Park, Ganesh Krishnan, Rohit Deep, Vijay Raghavendra</i>	193
Extracting Statistical Graph Features for Accurate and Efficient Time Series Classification <i>Daoyuan Li, Jessica Lin, Tegawendé Bissyandé, Jacques Klein, Yves Le Traon</i>	205
Counting Edges with Target Labels in Online Social Networks via Random Walk <i>Yang Wu, Cheng Long, Ada Fu, Zitong Chen</i>	217
An Homophily-based Approach for Fast Post Recommendation on Twitter <i>Quentin Grossetti, Camelia Constantin, Cedric du Mouza, Nicolas Travers</i>	229
Online Set Selection with Fairness and Diversity Constraints <i>Julia Stoyanovich, Ke Yang, H. Jagadish</i>	241
Apollo: Learning Query Correlations for Predictive Caching in Geo-Distributed Systems <i>Brad Glasbergen, Michael Abebe, Khuzaima Daudjee, Scott Foggo, Anil Pacaci</i>	253
Interactive Rule Refinement for Fraud Detection <i>Tova Milo, Slava Novgorodov, Wang-Chiew Tan</i>	265
Privacy Preserving Group Nearest Neighbor Search <i>Yuncheng Wu, Ke Wang, Zhilin Zhang, weipeng lin, Hong Chen, Cuiping Li</i>	277
Pattern Search in Temporal Social Networks <i>Andreas Züfle, Matthias Renz, Tobias Emrich, Maximilian Franzke</i>	289
Scalable and Dynamic Regeneration of Big Data Volumes <i>Anupam Sanghi, Raghav Sood, Jayant Haritsa, Srikanta Tirthapura</i>	301
TPStream: Low-Latency Temporal Pattern Matching on Event Streams <i>Michael Körber, Nikolaus Glombiewski, Bernhard Seeger</i>	313
QUASII: QUery-Aware Spatial Incremental Index <i>Mirjana Pavlovic, Darius Sidlauskas, Thomas Heinis, Anastasia Ailamaki</i>	325
Loom: Query-aware Partitioning of Online Graphs <i>Hugo Firth, Paolo Missier, Jack Aiston</i>	337
Kernel-Based Cardinality Estimation on Metric Data <i>Michael Mattig, Thomas Fober, Christian Beilschmidt, Bernhard Seeger</i>	349
GeoAlign: Interpolating Aggregates over Unaligned Partitions <i>Jie Song, Danai Koutra, Murali Mani, H. Jagadish</i>	361
Distributed query-aware quantization for high-dimensional similarity searches <i>Gheorghii Guzun, Guadalupe Canahuate</i>	373
Global-Scale Placement of Transactional Data Stores <i>Victor Zakhary, Faisal Nawab, Divy Agrawal, Amr El Abbadi</i>	385
SlickDeque: High Throughput and Low Latency Incremental Sliding-Window Aggregation <i>Anatoli Shein, Panos Chrysanthis, Alexandros Labrinidis</i>	397
Modeling and Exploiting Goal and Action Associations for Recommendations <i>Dimitra Papadimitriou, Yannis Velegrakis, Georgia Koutrika</i>	409
Short Papers	
Very-Low Random Projection Maps <i>Anastasios Zouzias, Michail Vlachos</i>	421

Interval Count Semi-Joins <i>Panagiotis Bouros, Nikos Mamoulis</i>	425
Notable Characteristics Search through Knowledge Graphs <i>Davide Mottin, Bastian Grasnck, Axel Kroschk, Patrick Siegler, Emmanuel Müller</i>	429
EmbedS: Scalable, Ontology-aware Graph Embeddings <i>Gonzalo Diaz, Achille Fokoue, Mohammad Sadoghi</i>	433
All that Incremental is not Efficient: Towards Recomputation Based Complex Event Processing for Expensive Queries <i>Abderrahmen Kammoun, Syed Gillani, Julien Subercaze, Stephane Frenot, Kamal Singh, Frederique Laforest, Jacques Fayolle</i>	437
DeepEye: Visualizing Your Data by Keyword Search <i>xuedi qin, Yuyu Luo, Nan Tang, Guoliang Li</i>	441
Research Directions in Blockchain Data Management and Analytics <i>Hoang Tam Vo, Ashish Kundu, Mukesh Mohania</i>	445
Scalable Active Temporal Constrained Clustering <i>Son Mai, Sihem Amer-Yahia, Ahlame Douzal Chouakria</i>	449
Global Range Encoding for Efficient Partition Elimination <i>Jeremy Chen, Reza Sherkat, Mihnea ANDREI, Heiko Gerwens</i>	453
NoFTL-KV: Tackling Write-Amplification on KV-Stores with Native Storage Management <i>Tobias Vincon, Sergej Hardock, Christian Riegger, Julian Oppermann, Andreas Koch, Ilia Petrov</i>	457
Towards Hypothetical Reasoning Using Distributed Provenance <i>Daniel Deutch, Yuval Moskovitch, Itay Polak, Noam Rinetzky</i>	461
On Answering Why-Not Queries Against Scientific Workflow Provenance <i>Khalid Belhajjame</i>	465
PRoST: Distributed Execution of SPARQL Queries Using Mixed Partitioning Strategies <i>Matteo Cossu, Michael Färber, Georg Lausen</i>	469
Deep Integration of Machine Learning Into Column Stores <i>Mark Raasveldt, Pedro Holanda, Hannes Mühleisen, Stefan Manegold</i>	473
Scalable Detection of Concept Drifts on Data Streams with Parallel Adaptive Windowing <i>Philipp Marian Grulich, Rene Saitenmacher, Jonas Traub, Sebastian Breß, Tilmann Rabl, Volker Markl</i>	477
Point-of-Interest Recommendation Using Heterogeneous Link Prediction <i>Alireza Pourali, Fattane Zarrinkalam, Ebrahim Bagheri</i>	481
MetisIDX - From Adaptive to Predictive Data Indexing <i>Elvis Teixeira, Paulo Amora, Javam Machado</i>	485
Efficient SIMD Vectorization for Hashing in OpenCL <i>Tobias Behrens, Viktor Rosenfeld, Jonas Traub, Sebastian Breß, Volker Markl</i>	489
Histogram Domain Ordering for Path Selectivity Estimation <i>Nikolay Yakovets, Li Wang, George Fletcher, Craig Taverner, Alexandra Poulouvassilis</i>	493
Nomadic Datacenters at the Network Edge: Data Management Challenges for the Cloud with Mobile Infrastructure <i>Faisal Nawab, Divy Agrawal, Amr El Abbadi</i>	497
Dynamic Resource Routing using Real-Time Information <i>Sebastian Schmoll, Matthias Schubert</i>	501

Data Structures for Efficient Computation of Influence Maximization and Influence Estimation <i>Diana Popova, Akshay Khot, Alex Thomo</i>	505
A Roadmap towards Declarative Similarity Queries <i>Nikolaus Augsten</i>	509
Tutorials	
Interactive Exploration of Composite Items <i>Sihem Amer-Yahia, Senjuti Basu Roy</i>	513
Recent Advances in Recommender Systems: Matrices, Bandits, and Blenders <i>Georgia Koutrika</i>	517
openCypher: New Directions in Property Graph Querying <i>Alastair Green, Martin Junghanns, Max Kiessling, Tobias Lindaaker, Stefan Plantikow, Petra Selmer</i>	520
Real-Time Data Management for Big Data <i>Wolfram Wingerath, Felix Gessert, Erik Witt, Steffen Friedrich, Norbert Ritter</i>	524
Industrial and Applications Papers	
Supporting Similarity Queries in Apache AsterixDB <i>Taewoo Kim, Wenhai Li, Alexander Behm, Inci Cetindil, Rares Vernica, Vinayak Borkar, Michael Carey, Chen Li</i>	528
L-Store: A Real-time OLTP and OLAP System <i>Mohammad Sadoghi, Souvik Bhattacharjee, Bishwaranjan Bhattacherjee, Mustafa Canim</i>	540
A Hybrid Approach for Alarm Verification using Stream Processing, Machine Learning and Text Analytics <i>Ana Sima, Kurt Stockinger, Katrin Affolter, Martin Braschler, Peter Monte, Lukas Kaiser</i>	552
Efficient Secure k-Nearest Neighbours over Encrypted Data <i>Manish Kesarwani, Akshar Kaul, Prasad Naldurg, Sikhar Patranabis, Gagandeep Singh, sameep mehta, Debdeep Mukhopadhyay</i>	564
A Parallel and Scalable Processor for JSON Data <i>Christina Pavlopoulou, E. Preston Carman Jr, Till Westmann, Michael Carey, Vassilis Tsotras</i>	576
An Automated System for Internet Pharmacy Verification <i>Alberto Cordioli, Themis Palpanas</i>	588
RQL: Retrospective Computations over Snapshot Sets <i>Nikos Tsikoudis, Liuba Shrira, Sara Cohen</i>	600
Big Data Analytics for Time Critical Mobility Forecasting: Recent Progress and Research Challenges <i>George Vouros, Akrivi Vlachou, Giorgos Santipantakis, Christos Doulkeridis, Nikos Pelekis, Harris Georgiou, Yannis Theodoridis, Kostas Patroumpas, Elias Alevizos, Alexander Artikis, Christophe Claramunt, Cyril Ray, David Scarlatti, Georg Fuchs, Gennady Andrienko, Natalia Andrienko, Michael Mock, Elena Camossi, Anne-Laure Jous-selme, Jose Manuel Cordero Garcia</i>	612
Scouter: A Stream Processing Web Analyzer to Contextualize Singularities <i>Badre Belabess, Musab Bairat, Jeeremy Lhez, Zakaria Khattabi, Yufan Zheng, Olivier CURE</i>	624
Finding Contrast Patterns for Mixed Streaming Data (Application track) <i>Rohan Khade, Jessica Lin, Nital Patel</i>	632
Demonstrations	
Recalibration of Analytics Workflows <i>Verena Kantere, Maxim Filatov, Maxim Filatov, Vasiliki Kantere, Verena Kantere</i>	642

Effective Quality Assurance for Data Labels through Crowdsourcing and Domain Expert Collaboration <i>Wei Lee, Chien-Wei Chang, Po-An Yang, Chi-Hsuan Huang, Ming-Kuang Wu, Chu-Cheng Hsieh, Kun-Ta Chuang</i>	646
Exploring Large Scholarly Networks with Hermes <i>Gabriel Campero Durand, Anusha Janardhana, Marcus Pinnecke, Yusra Shakeel, Jacob Krüger, Thomas Leich, Gunter Saake</i>	650
Don't write all data pages in one stream <i>Soyee Choi, Hyunwoo Park, Sang Won Lee</i>	654
eLinda: Explorer for Linked Data <i>Tal Yahav, Oren Kalinsky, Oren Mishali, Benny Kimelfeld</i>	658
FAIMUSS: Flexible Data Transformation to RDF from Multiple Streaming Sources <i>Giorgos Santipantakis, Apostolos Glenis, Nikolaos Kalaitzian, Akrivi Vlachou, Christos Doulkeridis, George Vouros</i>	662
SAMUEL: A Sharing-based Approach to processing Multiple SPARQL Queries with MapReduce <i>InA Kim, Kyong-Ha Lee, Kyuchul Lee</i>	666
GEDetector: Early Detection of Gathering Events Based on Cluster Containment Join in Trajectory Streams <i>Bin Zhao, Genlin Ji, Yu Yang, Zhaoyuan Yu, Xintao Liu, Ningfang Mi</i>	670
Reconciling Privacy and Data Sharing in a Smart and Connected Surrounding <i>Paul Tran-Van, Nicolas Anceaux, Philippe Pucheral</i>	674
Spatio-Temporal-Keyword Pattern Queries over Semantic Trajectories with Hermes@Neo4j <i>Fragkiskos Gryllakis, Nikos Pelekis, Christos Doulkeridis, Stylianos Sideridis, Yannis Theodoridis</i>	678
MDM: Governing Evolution in Big Data Ecosystems <i>Sergi Nadal, Alberto Abelló, Oscar Romero, Stijn Vansummeren, Panos Vassiliadis</i>	682
Provenance-Based Visual Data Exploration with EVLIN <i>Housseem BEN LAHMAR, Melanie Herschel, Michael Blumenschein, Daniel Keim</i>	686
Interactive Visualization of Large Similarity Graphs and Entity Resolution Clusters <i>M. Ali Rostami, Alieh Saeedi, Eric Peukert, Erhard Rahm</i>	690
FastOFD: Contextual Data Cleaning with Ontology Functional Dependencies <i>Zheng Zheng, Morteza Alipour, Zhi Qu, Ian Currie, Fei Chiang, Lukasz Golab, Jaroslaw Szlichta</i>	694
Analysis and Visualization of Urban Emission Measurements in Smart Cities <i>Dirk Ahlers, Frank Kraemer, Anders Braten, Xiufeng Liu, Fredrik Anthonisen, Patrick Driscoll, John Krogstie</i>	698
Pharos: Privacy Hazards of Replicating ORAM Stores <i>Victor Zakhary, Cetin Sahin, Amr El Abbadi, Huijia Lin, Stefano Tessaro</i>	702

ID Repair for Trajectories with Transition Graphs

Xingcan Cui[†] Xiaohui Yu^{§†} Xiaofang Zhou[‡] Jiong Guo[†]

[†]Shandong University, China [§]York University, Canada [‡]University of Queensland, Australia
 xccui@mail.sdu.edu.cn xhyu@yorku.ca zxf@itee.uq.edu.au jguo@sdu.edu.cn

ABSTRACT

In many surveillance applications, capture devices are set on fixed locations to track entities, leading to valuable spatio-temporal trajectories. However, sometimes the IDs of the entities in these trajectories are incorrectly identified due to various reasons (e.g., illumination conditions and partial occlusion). Since very often the movements of the entities are constrained by certain restrictions imposed by the application (e.g., vehicles must move along the given road network), we consider how to repair the erroneous IDs using transition graphs derived from such restrictions. Roughly speaking, the occurrence of erroneous IDs can cause a valid trajectory to be broken into trajectory fragments that violate some movement constraints imposed by the transition graph, and we aim to repair them by rewriting the IDs and merging the fragments. This problem is practically challenging since it is not easy to judge which IDs in the dataset are correct, and also there may be multiple candidates as the correct value for a single error. We formulate the repair process as an optimization problem and propose a two-phase repair paradigm, which includes candidate repair generation and compatible repair selection, to maximize the quality improvement estimated by a designed objective function. Though both phases are intractable, we propose effective algorithms to solve them through exploiting the locality and sparsity of trajectories. We further devise an index structure, as well as a pruning method to make the repair process more efficient. Experiments on both real and synthetic datasets demonstrate the effectiveness and efficiency of the proposed methods.

1 INTRODUCTION

Many surveillance related applications require the continuous tracking of entities over time in a specified area. For example, in maritime transport, surveillance devices can be set on ports to track the ships; in traffic surveillance systems, cameras are placed along city streets to capture images of passing vehicles. One of the main tasks for these applications is to identify the unique ID (which may be an atomic value or a composite one consisting of multiple features, such as name, color and shape) of each recorded entity (e.g., a ship or a vehicle) so that tracking records of those entities can be constituted.

For instance, vision-based algorithms are used to identify both the types [3] and names [15] of ships; similarly, optical character recognition (OCR) techniques are used to distill license plate numbers from the captured vehicle images. Due to various reasons (e.g., illumination conditions, partial occlusion or masking), it is not uncommon for the IDs of some entities to be incorrectly identified. Although much effort has been devoted to developing new techniques for improving the recognition accuracy, such errors are still unavoidable, especially when deliberate efforts are made

to prevent the entities from being recognized (e.g., in the case of smuggling at sea¹). According to recent studies [10, 15], the recognition rates of modern approaches are generally over 90% in lab environments, while in real-world settings, the rates may drop (e.g., to about 83% in the real traffic dataset we examined).

In this paper, we take a different perspective and aim to repair erroneous IDs by exploiting the inherent movement constraints, which are formally represented as *transition graphs*, that each entity must follow. Specifically, a transition graph is a directed graph with each vertex corresponding to a location and each edge a feasible move. Furthermore, some vertices are designated as the entrance/exit locations. In general, the transition graphs are the results of geographical restrictions (e.g., road networks), regulations (e.g., shipping routes), etc., and thus can be easily derived or sometimes even explicitly given.

1.1 A Motivating Example

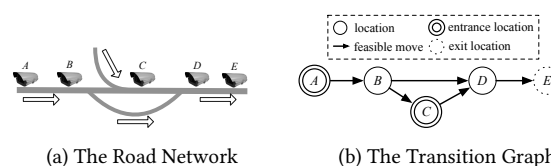


Figure 1: A Transition Graph Example

Example 1.1. As a running example, consider the transition graph depicted in Figure 1(b) for the road network shown in Figure 1(a) with surveillance cameras installed at locations A, B, \dots, E (where the hollow arrows indicate driving directions). The transition graph implies the following two movement constraints: (1) vehicles can only enter this area at locations A or C and leave from location E ; and (2) the move of a vehicle must match a directed edge in this graph.

Table 1 shows an example of the tracking records captured by the cameras in Figure 1(a). Without loss of generality, we assume that each tracking record contains at least three fields – the ID, the capture location and the capture timestamp. We also assume that errors occur only in the ID field, as the locations are fixed and the timestamps can be synchronized across cameras and are thus much less error-prone. Records with the same ID can be chronologically sorted and concatenated to form a trajectory. For convenience, we denote a trajectory by the ID followed by the sequence of locations, e.g., $GL21348\langle A \rightarrow B \rightarrow D \rightarrow E \rangle$ represents an entity with ID $GL21348$ moving from A to B to D to E .

Suppose that the dataset is complete, i.e., there are no missing records. Then each trajectory must satisfy both of the aforementioned movement constraints.

Example 1.2. Table 2 shows the composed trajectories from the tracking records in Table 1. Among the three trajectories, only the first one satisfies the movement constraints imposed by

¹<https://goo.gl/bKGvhh>

Table 1: Tracking Records

ID	Loc	Time
GL21348	A	08:09:10
GL21348	B	08:13:07
GL03245	C	08:17:23
GL21348	D	08:19:13
GL83248	D	08:19:40
GL21348	E	08:21:29
GL83248	E	08:21:30

Table 2: Trajectories

No.	Trajectory
1	GL21348(A → B → D → E)
2	GL03245(C)
3	GL83248(D → E)

the transition graph in Figure 1(b) and is thus considered valid; the second and third trajectories are invalid as they fail to satisfy the first movement constraint in Example 1.1.

Note that ID misidentification can cause “fracture” of a valid trajectory and therefore may render the trajectory invalid with respect to the given transition graph. The transition graph is distinct from most existing constraints [9, 25, 27], in that even trajectories with correct IDs may violate the constraints imposed by the graph and thus become invalid.

Example 1.3. Assume that the original trajectory for entity with ID GL83248 is GL83248(C → D → E). Unfortunately, its ID is misidentified as GL03245 by the camera at C. The trajectory is thus broken into the second and third trajectories in Table 2, both of which are invalid (though the second trajectory is actually error-free).

Some related approaches [27, 29] try to repair erroneous attributes in temporal events (e.g., logs from manufacturing) by exploiting the structural information or the neighborhood constraints of the activities. They propose efficient methods based on graph structures to detect dirty events and devise heuristic algorithms to repair them based on the *minimum change principle* [4]. However, these approaches fall short in our scenario mainly because (1) they perform isolated label rewritings while in our problem a single repair option may involve multiple ID-rewritings (as an ID may be identified as multiple erroneous values), (2) they do not consider the spatial relationships between the trajectories, which play an important role in our problem, and (3) the minimum change principle they follow may no longer be appropriate in our scenario.

1.2 The Present Work

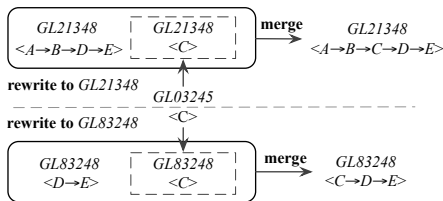


Figure 2: Two Repair Options for Trajectory GL03245[C]

We propose to repair the erroneous IDs through rewriting the IDs and merging the trajectory fragments to recover the “original” trajectories that are valid with respect to the transition graph.

Example 1.4. Consider the trajectory GL03245(C) in Table 2. As shown in Figure 2, by exploring the inherent location and timestamp relationships, we can rewrite the trajectory’s ID to GL83248 (or, GL21348) and then merge the corresponding tracking records chronologically to form a valid trajectory GL83248(C → D → E) (or GL21348(A → B → C → D → E)).

As shown in Figure 2, there could be multiple options to repair an invalid trajectory, which are mutually exclusive since it is logically inconsistent to repair a single ID to different values at the same time. As such, only one of the options can be used.

Various factors can be considered in evaluating the “goodness” of the two repair options in Example 1.4, e.g., the ID similarity and the number of invalid trajectories eliminated. In this example, the bottom repair option in Figure 2 (i.e., rewriting to GL83248) is more likely to be selected roughly because (1) compared with “GL21348”, the string “GL83248” is more similar to “GL03245” (in terms of edit distance), and (2) applying this option can eliminate all the invalid trajectories in the dataset, while applying the other one (i.e., rewriting to GL21348) will leave a dangling invalid trajectory GL83238(D → E) without other trajectories to merge with.

The examples above represent a simple case where the dataset contains only one misidentified ID. In practice, the problem of trajectory ID repair is much more complex because (1) it is non-trivial to judge which IDs in the dataset are correct, (2) generating the repair options, each of which may involve multiple trajectories, can be time consuming, and (3) there can be a large number of interrelated repair options and we must consider them as a whole to make global decisions.

To address these issues, we propose a repair paradigm that consists of two phases. In the first phase, we generate all potential repair options that meet certain criteria for later consideration; in the second phase, we use an evaluation function to estimate the data quality improvement brought by each repair option, and then search for a set of such options that can be applied in tandem to maximize an overall objective function (which reflects the global quality improvement for the given dataset). Specifically, we extract the core processes in these two phases as a clique generation problem and a weighted independent set problem, which are all NP-hard in general settings. To cope with the first problem, we add some restrictions on the cliques of interest and provide a backtracking algorithm. To solve the second one, we propose an approximate greedy algorithm by exploiting the probability of selecting a correct repair option. Furthermore, we also devise an index structure and a pruning method to make the whole repair approach more efficient.

In repairing the IDs, we do not invent new values and assume the correct IDs can always be found from the dataset, which is in line with most previous work on data repair or data matching in other settings [4, 22, 32]. While the datasets often exhibit another type of error, namely missing records, our focus in this paper is on repairing erroneous IDs, which constitute the main source of error in the datasets we have examined. The reason is that the problem of missing records are often mitigated through the deployment of other supporting or complementary technologies. For example, inductive-loop traffic detectors [2] installed at the same locations as the traffic cameras can detect almost all passing vehicles and trigger shots by the corresponding cameras. Therefore, it is rare for a vehicle not to be captured by the camera and its plate (either correctly or incorrectly) recognized. In general, dealing with missing records is a separate issue worthy of investigation in our future work; in fact, most existing methods on missing value recovery [30, 31] focus on the issue of missing records itself without tackling other data quality problems such as indistinguishable objects. Nonetheless, we conduct experiments in Section 6.3.3 to empirically evaluate the impact of missing records on the performance of the proposed methods.

1.3 Contributions

To the best of our knowledge, we are the first to study the problem of trajectory ID repair facilitated by the moving rules. In summary, we make the following contributions in this paper.

- (1) We propose a novel transition graph based trajectory ID repair problem, as well as a two-phase repair paradigm to solve it.
- (2) By exploiting the locality and sparsity of the spatio-temporal trajectories, we provide practical algorithms that can solve the problem effectively.
- (3) We further devise some optimization methods, which make our approach more efficient.
- (4) Extensive experiments are conducted on both real and synthetic datasets, which demonstrate the effectiveness and efficiency of the proposed methods.

The rest of the paper is organized as follows. In Section 2, we provide the preliminaries and formally define the problem. We present the two-phase repair paradigm and the detailed algorithms in Section 3 and Section 4, respectively. We further propose some optimization methods in Section 5, and show the experimental results in Section 6. We provide an overview of the related work in Section 7, and conclude this paper in Section 8.

2 PROBLEM DESCRIPTION

2.1 Preliminaries

We first define several terms that will be used throughout the paper.

Definition 2.1. Transition Graph, Entrance Location and Exit Location. A transition graph $\mathcal{G}_t = (\mathbf{V}, \mathbf{E})$ is a directed graph that represents the set of movement constraints that the entities must follow. Each vertex $loc \in \mathbf{V}$ represents a location (e.g., where a surveillance device is installed), and an edge $(loc_i, loc_j) \in \mathbf{E}$ indicates that an entity can directly move from location loc_i to location loc_j . Among these vertices (locations), there are some special ones from which the entities can enter or leave the area of interest. We call them the *entrance locations* (the set of which is denoted by \mathbf{I}) and *exit locations* (the set of which is denoted by \mathbf{O}).

For the transition graph shown in Example 1.1, $\mathbf{V} = \{A, B, C, D, E\}$, $\mathbf{E} = \{(A, B), (B, C), (B, D), (C, D), (D, E)\}$, $\mathbf{I} = \{A, C\}$ and $\mathbf{O} = \{E\}$.

Definition 2.2. Valid Path. Given a transition graph $\mathcal{G}_t = (\mathbf{V}, \mathbf{E})$ with the entrance location set \mathbf{I} and the exit location set \mathbf{O} , we call a location sequence $loc_1 \rightarrow loc_2 \cdots \rightarrow loc_q$ a *valid path* if it satisfies the following three conditions: (1) $loc_1 \in \mathbf{I}$, (2) $(loc_i, loc_{i+1}) \in \mathbf{E} (1 \leq i < q)$, and (3) $loc_q \in \mathbf{O}$.

Definition 2.3. Tracking Record. A tracking record r is a triple (id, loc, ts) , where id is the entity's unique identifier (which may be erroneous and is the subject of our study), loc is the location, and ts is the timestamp.

Definition 2.4. Trajectory, Valid Trajectory, and Invalid Trajectory. A trajectory T is a chronologically ordered sequence of tracking records with the same ID (denoted by $T.id$), i.e., $T = r_1 \rightarrow r_2 \cdots \rightarrow r_q$ with $T.id = r_1.id = r_2.id = \cdots = r_q.id$ and $r_1.ts < r_2.ts < \cdots < r_q.ts$. We can represent a trajectory with the ID followed by the location sequence, e.g., $GL12345\langle A \rightarrow B \rightarrow C \rangle$. Given a transition graph \mathcal{G}_t and a trajectory T , we call T a *valid trajectory* (or *VT* for short) if the location sequence $r_1.loc \rightarrow$

$r_2.loc \cdots \rightarrow r_q.loc$ of T is a valid path w.r.t. \mathcal{G}_t . Otherwise we call T an *invalid trajectory* (or *IVT* for short).

In an ideal setting, each trajectory in a dataset is error-free and contains all the tracking records of an entity (e.g., $GL21348\langle A \rightarrow B \rightarrow D \rightarrow E \rangle$ in Example 1.2). Due to ID errors, however, a trajectory may be broken into multiple fragments, each containing tracking records with a different ID (e.g., $GL03245\langle C \rangle$ and $GL83248\langle D \rightarrow E \rangle$ in Example 1.2). Certainly, at most one of the IDs is correct, and the rest are all erroneous. Most of the time, those fragments are invalid trajectories (including the one with the correct ID), but there does exist a slight possibility that each of the fragments coincidentally corresponds to a valid path in the transition graph and is thus deemed valid. Considering its rarity, we ignore this case in our subsequent discussion.

We view the ID repair problem as one of "restoring" the original true trajectory through ID rewriting. That is, we seek to find a subset of trajectories and rewrite their IDs to (hopefully) the correct one, and then merge those trajectories to form a valid trajectory. As such, we introduce the following definitions.

Definition 2.5. Join, Joinable Subset and Target ID. Given a transition graph \mathcal{G}_t , we define *join* on a trajectory set \mathcal{T}' and an existing ID r from \mathcal{T}' as first rewriting the ID for each $T \in \mathcal{T}'$ to r and then merging all tracking records in \mathcal{T}' chronologically to constitute a new trajectory \widehat{T}_r , i.e., $\widehat{T}_r = join(\mathcal{T}', r)$. The join is valid iff the newly formed trajectory \widehat{T}_r is a VT w.r.t. \mathcal{G}_t , and we call r the *target ID* and \mathcal{T}' a *joinable subset* iff such a valid join exists.

Definition 2.6. Candidate Repair. A candidate repair (or *repair* for short) R is a pair (\mathcal{T}', r) consisting of a joinable subset \mathcal{T}' and a target ID r . For a repair R , the corresponding joinable subset and the invalid trajectories contained therein are denoted by $jns(R)$ and $ivt(R)$. With R defined, the join operation can be rephrased as $\widehat{T}_r = join(R)$. If r is the true ID for all trajectories contained in \mathcal{T}' (which implies that they actually come from the same entity), and \widehat{T}_r is the true trajectory for the entity with ID r , we call R a *correct candidate repair* (or *correct repair* for short). For two candidate repairs R_i and R_j , if their joinable subsets are mutually exclusive, i.e., $jns(R_i) \cap jns(R_j) = \emptyset$, we say that R_i and R_j are *compatible*; otherwise *incompatible*. If all pairs of repairs in a set of repairs are compatible, then we call this set a *compatible repair set*.

Figure 2 shows two candidate repairs whose correctness is unknown. They are incompatible due to the sharing of trajectory $GL03245\langle C \rangle$.

Given a trajectory set \mathcal{T} and a compatible repair set \mathcal{R}' , a new trajectory set $\widehat{\mathcal{T}}$ can be produced by joining the trajectories indicated by each $R \in \mathcal{R}'$. As illustrated before, the reason why the repairs must be compatible is that it does not make sense to rewrite a trajectory's ID to more than one target ID at the same time.

2.2 Problem Statement

Given a transition graph \mathcal{G}_t , a set of trajectories \mathcal{T} with ID errors, we use \mathcal{R} to represent the set which contains all the potential candidate repairs. The ID repair problem can be regarded as searching a compatible repair set $\mathcal{R}' \subseteq \mathcal{R}$, and joining trajectories designated by the compatible candidate repairs in \mathcal{R}' to obtain a new trajectory set $\widehat{\mathcal{T}}$.

Note that there may exist various (incompatible) candidate repairs in \mathcal{R} for a single trajectory, and our goal is to find the

most promising one. We thus need a function $\omega(R)$ to evaluate the effectiveness of R , which serves as an estimate of how much quality improvement can be gained by R .

2.2.1 The Evaluation Function for Repair Effectiveness. We first discuss the factors that we have considered in devising a suitable evaluation function for the effectiveness of a repair.

- **The individual fitness of a repair.** In real applications, the erroneous IDs often bear some similarities with their correct values, i.e., the more similar two IDs are, the more likely they correspond to the same entity. In the case of composite IDs, even if attempts are made to camouflage the entities with a fake name, the remaining components of the IDs, such as color and type, are more difficult to conceal and thus the erroneous IDs would still be similar to the true IDs. For that reason, we use ID similarity to evaluate the individual fitness of a repair. This similarity can be measured by the distance between the strings or feature vectors representing the IDs, and there have been dozens of metrics (e.g., edit distance, overlap coefficient, and cosine similarity) proposed in the literature for this purpose [24]. In this paper, we choose edit distance as the similarity metric, but this can be replaced by other metrics for different applications. Specifically, for a repair $R = (\mathcal{T}', r)$, we use the *similarity function* $\text{sim}(R)$, which is based on the minimum similarity from an ID in \mathcal{T}' to the target ID r , to evaluate the individual fitness of R :

$$\text{sim}(R) = \min_{T \in \mathcal{T}'} \left(1 - \frac{\text{dist}(r, T.id)}{\max(|r|, |T.id|)} \right) \quad (1)$$

where $\text{dist}(r, T_i.id)$ is the edit distance between r and $T_i.id$, and $|T_i.id|$ is the length of $T_i.id$. Apparently, the range of the similarity function is $[0, 1]$.

- **The potency of a repair.** As illustrated before, ID errors will cause invalid trajectories. Candidate repairs that fix more IVTs are considered more “powerful” and are expected to bring greater quality improvement to a dataset.
- **The rarity of a repair.** For a set of trajectories \mathcal{T} and a set of corresponding potential candidate repairs \mathcal{R} , each IVT $T' \in \mathcal{T}$ may be covered by multiple repairs in \mathcal{R} . We call the number of those repairs the *degree of T'* (denoted by $d(T')$). A smaller degree implies that the trajectory is more “endangered”, i.e., there are fewer candidate repairs that are able to fix it. On the other hand, each such candidate is considered more precious or rarer and thus should be preferred. We define the *rarity* of a repair R as

$$ra(R) = \min_{T \in \text{ivt}(R)} d(T) \quad (2)$$

The value of $ra(R)$ ranges from 1 to $|\mathcal{R}|$. When $ra(R)$ takes the minimum value of 1, it means that only R can fix a certain IVT in the dataset.

Different effectiveness evaluation functions can be designed based on the factors above. We find the following one performs well in our scenarios:

$$\omega(R) = \begin{cases} 0 & |\text{ivt}(R)| = 0 \\ \text{sim}(R) + \lambda \log_{ra(R)+1} |\text{ivt}(R)| & |\text{ivt}(R)| \geq 1 \end{cases} \quad (3)$$

where $\text{sim}(R)$ and $ra(R)$ are the similarity function and the rarity function respectively, $|\text{ivt}(R)|$ represents the number of invalid trajectories in R , and $\lambda \in (0, 1)$ is a coefficient controlling the trade-off between the two terms.

In Equation (3), the first term $\text{sim}(R)$ acts as assurance on the matching fitness and makes it unlikely for an ID to be rewritten to another arbitrary ID. The second term $\log_{ra(R)+1} |\text{ivt}(R)|$ represents the potency impact scaled by the rarity factor. According to this term, repairs holding more invalid trajectories and being more rare will be more effective. In general, $ra(R) + 1 \gg |\text{ivt}(R)|$ and thus the range for $\log_{ra(R)+1} |\text{ivt}(R)|$ is also $[0, 1]$.

Note that due to the introduction of $ra(R)$, the effectiveness of a repair cannot be evaluated unless a full candidate repair set \mathcal{R} is provided.

2.2.2 The ID Repair Problem. Given a trajectory set \mathcal{T} (with ID errors), a transition graph \mathcal{G}_t , and an evaluation function ω for repair effectiveness, the ID repair problem is to search for a compatible repair set \mathcal{R}' that can maximize the sum of the repair effectiveness. Formally, this can be described as

$$\begin{aligned} & \underset{\mathcal{R}'}{\text{maximize}} && \Omega(\mathcal{R}') = \sum_{R \in \mathcal{R}'} \omega(R) \\ & \text{subject to} && jns(R_i) \cap jns(R_j) = \emptyset, \forall R_i, R_j \in \mathcal{R}'. \end{aligned} \quad (4)$$

2.3 Applicability and Assumptions

The ID repair approach we adopt exploits the locality and sparsity properties of the real world spatio-temporal trajectories:

- **Locality of movement** – a given entity is more likely to move within a local geospatial neighborhood than “jumping” between far-away locations in a relatively short time span. This implies that it is more likely to find the true value of an erroneous ID based on entities that are close in time and space.
- **Sparsity of IDs** – two entities with different but very similar IDs are highly unlikely to appear in the same local neighborhood during a relatively short period of time.

Based on these properties, we make the following assumptions. First, we assume that identical IDs in the data, whether correct or not, belong to the same entity. In other words, we would not break a trajectory into smaller pieces. In exceptional cases, it may so happen that a tracking record with an erroneous ID becomes part of a valid trajectory in place of a missing record with the true ID; but such cases are so rare due to the sparsity of IDs that we could consider them negligible.

Second, we consider all records with the same ID constitute a trajectory with bounded length and time span. The rationale is that despite some entities intending to wander around in the area of interest, the majority should be passing traffic and thus their trajectories should not be too long. Based on this assumption, we set two bounds θ and η , where θ is the maximum possible length of a VT (i.e., the maximum number of tracking records in it) in a dataset, and η is the maximum time span for a VT.

Finally, we assume that the error rate for ID identification is not too high (which is consistent with what we observe from real data), i.e., the number of trajectory fragments caused by erroneous IDs in a trajectory is limited. We use a bound ζ to represent the maximum possible number of trajectories in a joinable subset. Although there may be extraordinary cases where these assumed bounds do not hold, their establishment grants us the opportunity to significantly improve the efficiency of the proposed algorithms, as will be shown in Section 6.2.

3 A TWO-PHASE REPAIR PARADIGM

We now describe a two-phase repair paradigm that serves as the framework for solving the ID repair problem. The detailed

algorithms involved in this paradigm will be presented in Section 4.

3.1 Overview of the Paradigm

The basic idea of the proposed paradigm is to first generate all possible candidate repairs and then select a set of repairs that can maximize the objective function (Equation (4)). Figure 3 depicts the repair paradigm consisting of two phases: *candidate repair generation* and *compatible repair selection*.

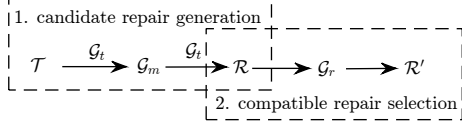


Figure 3: The Two-Phase Repair Paradigm

In the candidate repair generation phase, we generate all potential candidate repairs \mathcal{R} (which are not necessarily compatible with each other) from the input trajectory set \mathcal{T} according to the transition graph \mathcal{G}_t . This phase can be accomplished using an undirected graph with each vertex representing a different trajectory and each edge indicating that the two trajectories corresponding to its connected vertices can appear in the same joinable subset. We call this graph the *trajectory graph* (denoted by \mathcal{G}_m).

In the compatible repair selection phase, we perform the actual ID repair by selecting a set of compatible repairs $\mathcal{R}' \subseteq \mathcal{R}$ that have the maximum total effectiveness in terms of Equation (4). This task can be carried out by introducing another undirected graph that reflects the incompatible relationships between different repairs in \mathcal{R} . We call this graph the *repair graph* (denoted by \mathcal{G}_r).

3.2 Candidate Repair Generation

This phase performs two core tasks, namely *joinable subset determination* and *target ID assignment*.

3.2.1 Joinable subset determination. Before delving into the details of joinable subset determination, we first introduce two predicates.

- (1) **The *cex* predicate.** Given a transition graph \mathcal{G}_t , the *cex* predicate works by checking whether two trajectories can coexist in a joinable subset w.r.t. \mathcal{G}_t , i.e., $\{\mathbf{T}_x, \mathbf{T}_y\} | cex(\mathbf{T}_x, \mathbf{T}_y) = \{\mathbf{T}_x, \mathbf{T}_y\} \exists \mathcal{T}' (\mathbf{T}_x, \mathbf{T}_y \in \mathcal{T}')$, and \mathcal{T}' is a joinable subset. Apparently, only if the location sequence for the chronologically merged records of the two trajectories is a subsequence (which does not have to be continuous) of a path in \mathcal{G}_t , can this predicate evaluate to true.
- (2) **The *jnb* predicate.** Given a transition graph \mathcal{G}_t , the *jnb* predicate is used to determine whether a set of trajectories is a joinable subset w.r.t. \mathcal{G}_t , i.e., $\{\mathcal{T}_x\} | jnb(\mathcal{T}_x) = \{\mathcal{T}_x\} \exists \mathcal{T}' (\mathcal{T}_x = \mathcal{T}')$, and \mathcal{T}' is a joinable subset. This can be performed by checking whether the location sequence for the chronologically merged records is a valid path in \mathcal{G}_t . As a special case, for a trajectory set with only one element, the *jnb* predicate evaluates to *true* if that element is a valid trajectory.

We first use the *cex* predicate to construct the trajectory graph \mathcal{G}_m . Specifically, each vertex v_i in \mathcal{G}_m corresponds to a $\mathbf{T}_i \in \mathcal{T}$, and each undirected edge (v_i, v_j) corresponds to a trajectory pair $(\mathbf{T}_i, \mathbf{T}_j)$ such that *cex*($\mathbf{T}_i, \mathbf{T}_j$) evaluates to true.

Example 3.1. Let us use $\mathbf{T}_1, \mathbf{T}_2$, and \mathbf{T}_3 to represent the three trajectories shown in Table 2. To construct \mathcal{G}_m , we first create

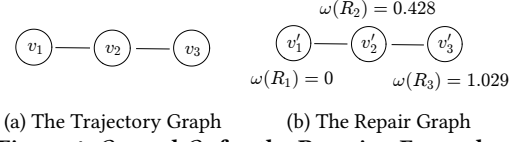


Figure 4: \mathcal{G}_m and \mathcal{G}_r for the Running Example

three vertices v_1, v_2 , and v_3 , corresponding to $\mathbf{T}_1, \mathbf{T}_2$, and \mathbf{T}_3 respectively. Since both *cex*($\mathbf{T}_1, \mathbf{T}_2$) and *cex*($\mathbf{T}_2, \mathbf{T}_3$) evaluate to true, two edges (v_1, v_2) and (v_2, v_3) are added to \mathcal{G}_m . Figure 4(a) shows the constructed trajectory graph for the dataset.

Such construction allows the problem of searching for joinable subsets to be transformed to operations on \mathcal{G}_m , as shown in the theorem below.

THEOREM 3.2. *A necessary but not sufficient condition for a trajectory set to be a joinable subset is that its corresponding vertex set in \mathcal{G}_m is a clique.*

To compute the joinable subsets, we first generate all cliques in \mathcal{G}_m and then use the *jnb* predicate to check whether their corresponding trajectory sets can really be joinable subsets. Actually, the clique generation process can be regarded as a preprocessing step which saves us from enumerating all the trajectory combinations. In general settings, listing all cliques in a graph is NP-hard [19]. Fortunately, since both the length of a VT and the number of trajectories contained in a candidate repair are bounded according to our aforementioned assumptions, the problem can be solved in polynomial time in our case. We present the detailed algorithm for generating the qualified-cliques in Section 4.1.2.

Example 3.3. For the trajectory graph shown in Figure 4(a), there are five cliques: $\{v_1\}, \{v_2\}, \{v_3\}, \{v_1, v_2\}$, and $\{v_2, v_3\}$. However, evaluating *jnb* on their corresponding trajectory sets reveals that there are only three joinable subsets: $\{\mathbf{T}_1\}, \{\mathbf{T}_1, \mathbf{T}_2\}$, and $\{\mathbf{T}_2, \mathbf{T}_3\}$.

3.2.2 Target ID Assignment. After generating all the joinable subsets, we assign target IDs to them. Given a joinable subset \mathcal{T}' , the target ID is decided by selecting a trajectory \mathbf{T}_c that maximizes the following equation.

$$\mathbf{T}_c = \underset{\mathbf{T}_i \in \mathcal{T}'}{\operatorname{argmax}} \left(\sum_{\mathbf{T}_j \in \mathcal{T}'} \frac{|\mathbf{T}_i|}{|\mathbf{T}_j|} \left(1 - \frac{\operatorname{dist}(\mathbf{T}_i.\operatorname{id}, \mathbf{T}_j.\operatorname{id})}{\max(|\mathbf{T}_i.\operatorname{id}|, |\mathbf{T}_j.\operatorname{id}|)} \right) \right) \quad (5)$$

In this equation, $|\mathbf{T}_i|$ ($|\mathbf{T}_j|$) is the length of trajectory \mathbf{T}_i (\mathbf{T}_j), $|\mathbf{T}_i.\operatorname{id}|$ ($|\mathbf{T}_j.\operatorname{id}|$) is the length of ID $\mathbf{T}_i.\operatorname{id}$ ($\mathbf{T}_j.\operatorname{id}$), and $\operatorname{dist}(\mathbf{T}_i.\operatorname{id}, \mathbf{T}_j.\operatorname{id})$ is the edit distance between them. The rationale behind the choice of this equation is that when all trajectories have the same length, our goal is to choose a target ID that can maximize the sum of similarities of all IDs with the target ID. We also give preference to longer trajectories, as the error rate for ID identification is usually low and it is unlikely for the same error to be made at consecutive locations in a trajectory. Note that we choose to use edit distance here to measure the (dis-)similarity between IDs, but other distance measures can also be used where appropriate.

Example 3.4. Based on Equation (5), we assign *GL21348*, *GL21348*, and *GL83248* as the target IDs for the three joinable subsets $\{\mathbf{T}_1\}, \{\mathbf{T}_1, \mathbf{T}_2\}$, and $\{\mathbf{T}_2, \mathbf{T}_3\}$ generated in Example 3.3 and combine them respectively to generate three candidate repairs: $R_1 = (\{\mathbf{T}_1\}, \text{GL21348})$, $R_2 = (\{\mathbf{T}_1, \mathbf{T}_2\}, \text{GL21348})$, and $R_3 = (\{\mathbf{T}_2, \mathbf{T}_3\}, \text{GL83248})$. Then we calculate their effectiveness values according to Equation (3), which are $\omega(R_1) = 0$, $\omega(R_2) = 0.428$, and $\omega(R_3) = 1.029$.

3.3 Compatible Repair Selection

In the previous phase, we have generated all candidate repairs \mathcal{R} . Next, we select compatible repairs from \mathcal{R} to maximize the objective function in Equation (4). The selection process can be mapped to operations on another undirected graph, namely the repair graph, \mathcal{G}_r , which can be constructed as follows: (1) for each candidate repair $R_i \in \mathcal{R}$, add a corresponding vertex v'_i to \mathcal{G}_r ; (2) if $R_i, R_j \in \mathcal{R}$ share an identical trajectory, add an undirected edge (v'_i, v'_j) to \mathcal{G}_r .

Example 3.5. To construct \mathcal{G}_r for the three candidate repairs generated in Example 3.4, we first add three corresponding vertices v'_1, v'_2 , and v'_3 . Since $jns(R_1) \cap jns(R_2) = \{T_1\}$ and $jns(R_2) \cap jns(R_3) = \{T_2\}$, two edges (v'_1, v'_2) and (v'_2, v'_3) are added. Finally, we get the repair graph shown in Figure 4(b).

Such construction allows us to view the problem of selecting compatible repairs as packing vertices from \mathcal{G}_r where no pairs are adjacent. This translates to the well known *weighted independent set problem*, which is NP-hard in common settings [23]. Considering the inherent relationships between repairs, we present a greedy algorithm to approximately solve this problem in Section 4.2.

4 ALGORITHMS

In this section, we present the core algorithms for the two-phase repair paradigm.

4.1 Algorithms for Repair Generation

4.1.1 Evaluating the cex and jnb predicates. Given a transition graph \mathcal{G}_t , the *cex* predicate determines whether two trajectories T_1 and T_2 can coexist in a joinable subset. The key idea in evaluating this predicate is to check whether the location sequence for the chronologically merged records is a subsequence of a path in \mathcal{G}_t . This can be considered a reachability problem, i.e., for the merged location sequence $loc_1 \rightarrow loc_2 \rightarrow \dots \rightarrow loc_q$ ($q = |T_1| + |T_2|$), if loc_i and loc_{i+1} belong to different trajectories, we check whether loc_{i+1} is reachable from loc_i .

A straightforward solution for this problem using breadth-first (or depth-first) search takes linear time, but we can do some pre-processing to have it done in constant time. Specifically, the Floyd Warshall algorithm [16] can be employed to calculate the shortest-path matrix M for \mathcal{G}_t , where $M[i][j]$ indicates the number of edges in the shortest path from loc_i to loc_j . After this preprocessing step, the reachability queries can be answered instantly by consulting the elements in M .

Moreover, recall that we have two user-defined bounds, the maximum length θ and the maximum time span η for each *VT*. Thus we should also check whether $|T_1| + |T_2| \leq \theta$ and whether the time span for the merged sequence exceeds η . Putting things together, we show the algorithm for evaluating the *cex* predicate in Algorithm 1.

Compared with the *cex* predicate, the *jnb* predicate is more strict in that it evaluates to *true* only if the given trajectories can perfectly make up a joinable subset. The algorithm for this predicate is similar to that for the *cex* predicate with the following two additional restrictions: (1) the location attributes in the earliest and the latest records of the input trajectories must be an entrance location and an exit location in \mathcal{G}_t , respectively, and (2) no matter whether two adjacent records (in the merged sequence) r_i and r_{i+1} belong to the same trajectory or not, there must be an

Algorithm 1 Algorithm for the *cex* predicate

Input: The reachability matrix M for \mathcal{G}_t ; the maximum length θ ; the maximum time span η ; two trajectories T_1 and T_2 .

Output: *true* if T_1 and T_2 can coexist in a joinable subset; *false* otherwise.

```

1: if  $|T_1| + |T_2| > \theta$  then
2:   return false
3: merge records in  $T_1$  and  $T_2$  by their timestamps to get a
   sequence  $r_1 \rightarrow r_2 \rightarrow \dots \rightarrow r_q$ ;
4: if  $r_q.ts - r_1.ts > \eta$  then
5:   return false
6: for all  $r_i$  and  $r_{i+1}$  in the merged sequence do
7:   if  $r_i.id \neq r_{i+1}.id$  then
8:     if  $M[i][i+1] \geq \theta$  then
9:       return false
10: return true

```

edge from $r_i.loc$ to $r_{i+1}.loc$. Due to space limitation, the detailed algorithm for evaluating the *jnb* predicate is omitted.

4.1.2 Generating Qualified Cliques. We now introduce the algorithm for generating the qualified cliques in \mathcal{G}_m . In general, enumerating all cliques in an undirected graph requires exponential time, and the running time is output-sensitive (i.e., the running time depends on the size of the output). However, when the trajectory length and clique size are bounded (by θ and ζ respectively), the problem can be much simplified. We show how to generate the qualified cliques in Algorithm 2, which runs in $O(|V_m|^\zeta)$ time.

Algorithm 2 Algorithm for generating qualified cliques

Input: The adjacency matrix representation of trajectory graph $\mathcal{G}_m = \{V_m, E_m\}$; a set C to store vertices in the current clique; an index list L for vertices; the maximum trajectory length θ ; the maximum clique size ζ .

Output: a set of generated cliques *results*.

```

1: fill  $L$  with  $1, 2, \dots, |V_m|$ 
2: function CLIQUE( $C, L$ )
3:   for  $i \leftarrow L.size(), 1$  do
4:      $v \leftarrow L.get(i)$ 
5:      $C \leftarrow C \cup \{T_v\}$ 
6:     create a new list  $L_{new}$ 
7:     for  $j \leftarrow 0, i$  do
8:        $w \leftarrow L.get(j)$ 
9:       if  $(v, w) \in E_m$  then
10:         $L_{new}.add(w)$ 
11:         $results \leftarrow results \cup \{C\}$ 
12:        if  $\neg L_{new}.empty()$  and  $C.RecordNumber < \theta$  and
            $|C| < \zeta$  then
13:          CLIQUE( $C, L_{new}$ )
14:           $C \leftarrow C \setminus \{T_v\}$ 
15:           $L.remove(i)$ 
16: return results

```

The main idea of the generation algorithm is backtracking. It iteratively adds to a temporary vertex set C a vertex that is adjacent to all existing ones in C from an input vertex list, outputs the result, starts a recursion with a new list of vertices that are adjacent to the newly added vertex, and finally removes the vertex added in this round. As shown in Line 12, unnecessary recursions are eliminated using the bounds θ and ζ .

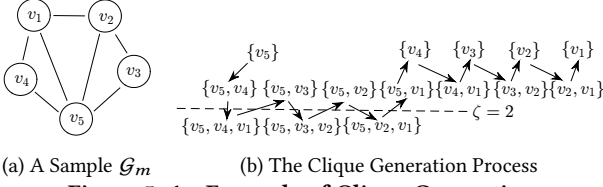


Figure 5: An Example of Clique Generation

Example 4.1. Figure 5 shows an example of the clique generation process. Given a trajectory graph with 5 vertices, the 15 cliques are generated in turn with Algorithm 2. As shown in Figure 5(b), if $\zeta = 2$, the algorithm will automatically skip generating cliques beneath the dashed line.

4.2 Algorithms for Repair Selection

As discussed in Section 3.3, the repair selection problem corresponds to a weighted-independent set problem on \mathcal{G}_r , which is NP-hard. In search of efficient solutions, consider the effectiveness evaluation function in Equation (4). It is defined as an indicator of the potential quality improvement due to a repair, but by no means a definitive measure of the true improvement. That is, for two compatible repair sets \mathcal{R}'_1 and \mathcal{R}'_2 , if $\Omega(\mathcal{R}'_1) > \Omega(\mathcal{R}'_2)$, it just indicates that \mathcal{R}'_1 is likely better than \mathcal{R}'_2 , but not definitely, especially when the values of $\Omega(\mathcal{R}'_1)$ and $\Omega(\mathcal{R}'_2)$ are close. This is confirmed by a large number of experiments on different datasets, from which we find that the Ω values of the optimal compatible repair sets (which include all the correct repairs) are randomly distributed in the proximity of, but not exactly the same as, the optimal results from the weighted-independent set problem.

The observation inspires us to seek approximate solutions to the repair selection problem instead. Many heuristic algorithms have been proposed for the weighted-independent set problem (see [5] for a survey). Here we propose a greedy algorithm named *maximum-effectiveness first* (EMAX), which gives superior empirical results in our settings. As shown in Algorithm 3, the EMAX algorithm always selects from \mathcal{G}_r a vertex whose corresponding repair is evaluated to be the most effective according to Equation (3), and then discards its adjacent vertices until there is no vertex left.

Algorithm 3 The EMAX algorithm

Input: the repair graph $\mathcal{G}_r = (\mathbf{V}_r, \mathbf{E}_r)$.

Output: a set of selected vertices \mathcal{V} .

- 1: sort vertices in \mathbf{V}_r by the ω values of their corresponding repairs in a decreasing order
- 2: **for all** v in \mathbf{V}_r **do**
- 3: **if** $v.discard = false$ **then**
- 4: add v to \mathcal{V}
- 5: **for all** v_a adjacent to v **do**
- 6: $v_a.discard \leftarrow true$
- 7: **return** \mathcal{V}

Compared with the exact algorithm that requires exponential running time, the EMAX algorithm runs in only $O(|\mathbf{V}_r| \log |\mathbf{V}_r|)$ time. The rationale behind this heuristic is that, repairs evaluated to be more effective are more likely to be correct, and thus giving them priorities makes it more likely to select the best result.

Example 4.2. For the repair graph shown in Figure 4(b), the EMAX algorithm first selects the vertex v'_3 corresponding to candidate repair R_3 (since it is the most effective according to Equation (3)) and then discards v'_2 adjacent to v'_3 . Since the effectiveness of R_1 , the only vertex left, is zero, v'_1 will not be selected.

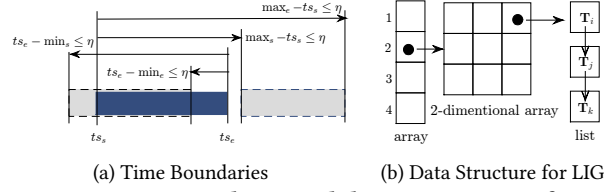


Figure 6: Time Boundaries and the Data Structure for LIG

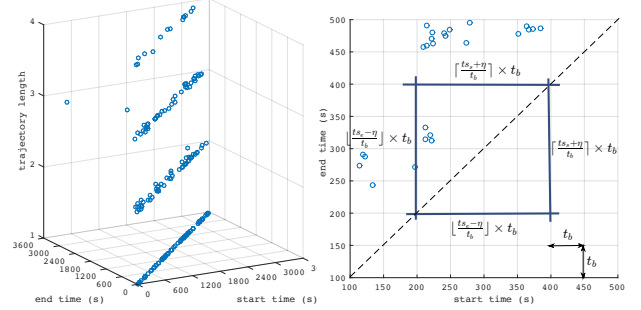


Figure 7: Overview of the Length-Indexed Grids

5 OPTIMIZATION

In this section, we provide some optimization methods to make the repair approach more efficient.

5.1 An Index for Constructing \mathcal{G}_m

The candidate repair generation phase requires evaluating the *ce χ* predicate on each pair of trajectories in \mathcal{T} for constructing the trajectory graph \mathcal{G}_m . Suppose that there are $|\mathbf{V}_m|$ trajectories. This procedure requires $O(|\mathbf{V}_m|^2)$ comparisons, which could be costly in practice. Recall that valid trajectories are upper-bounded in length and time span, and we thus make use of the bounds θ and η to filter out some unnecessary comparisons.

Definition 5.1. Start Time and End Time. The *start time* and the *end time* of a trajectory \mathbf{T} are defined as the timestamps of the earliest and latest records in \mathbf{T} , respectively.

Given a trajectory \mathbf{T}_k with start time ts_s and end time ts_e , another trajectory \mathbf{T}_u that may constitute a joinable subset with \mathbf{T}_k must first meet the length criterion, i.e., $|\mathbf{T}_u| \leq \theta - |\mathbf{T}_k|$. Also, for the bound on time span η , the max/min start time (denoted by max_s and min_s) and max/min end time (denoted by max_e and min_e) of \mathbf{T}_u should satisfy the following inequalities (which are demonstrated in Figure 6(a)): $ts_e - min_s \leq \eta$, $max_s - ts_s \leq \eta$, $ts_e - min_e \leq \eta$, and $max_e - ts_s \leq \eta$. According to these inequalities, both the start time and end time of \mathbf{T}_u should fall in $[ts_e - \eta, ts_s + \eta]$, and we can transform the criteria into a range query on a three-dimensional index structure on the trajectories called Length-Indexed Grids (LIG).

Overview. As shown in Figure 7(a), the three dimensions of LIG are the length, the start time and the end time of a trajectory. Specifically, we divide the time span of interest along both the start time and end time dimensions into time bins with fixed size t_b , resulting in a two-dimensional time grid shown in Figure 7(b). A separate time grid is created for each trajectory length appearing in the dataset.

The Data Structure. Figure 6(b) illustrates the data structure of LIG. An array is used to store the grids with different trajectory lengths. Each grid is actually a two-dimensional array. Trajectories are distributed to grids according to their start/end times and trajectories in the same grid are linked to be an element of

the two-dimensional array. We construct LIG by successively add trajectories. For each trajectory, we first decide the grid it belongs to according to the trajectory’s length. Then we assign a time grid for the trajectory and add it as a new element to the tail of the corresponding list.

Usage. We use the index to answer the range query by first deciding a set of feasible grids according to the trajectory’s length and the threshold θ . After that, in each grid, we select trajectory lists that meet the start/end time restrictions from the two-dimensional array. Without loss of generality, suppose that the timestamps of tracking records are represented as offsets to the earliest timestamp in the dataset. Then the target trajectories we are interested in should be contained in elements whose indices are bounded by $[\lfloor \frac{ts_s - \eta}{t_b} \rfloor \times t_b, \lceil \frac{ts_e + \eta}{t_b} \rceil \times t_b]$ in both dimensions.

With the index technique provided above, we can prune many useless trajectory comparisons. As the time grids are static, the index can be constructed efficiently in $\Theta(|V_m|)$ time. As such, the running time for generating \mathcal{G}_m can be significantly reduced.

5.2 A Pruning Method for Clique Generation

In Section 4.1.2, we show how to generate qualified cliques from the trajectory graph \mathcal{G}_m . All the trajectory sets corresponding to the cliques will be further checked by the *jnb* predicate to see if they are really joinable subsets. Considering that Algorithm 2 is output-sensitive, it will be more efficient if we can eliminate some worthless vertex combinations early on during the clique generation process. We propose an optimization method named *minimum cover prefix pruning* for this purpose.

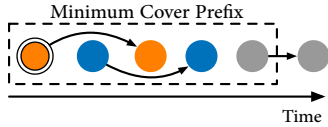


Figure 8: The Minimum Cover Prefix

Definition 5.2. Minimum Cover Prefix. As shown in Figure 8, given a trajectory set $\mathcal{T} = \{T_1, T_2, \dots, T_p\}$, we can merge their tracking records chronologically to get a sequence $r_1 \rightarrow r_2 \rightarrow \dots \rightarrow r_q$. The *minimum cover prefix* (abbreviated as MCP) for \mathcal{T} is defined as the minimum prefix of this sequence that contains at least one tracking record from all trajectories in \mathcal{T} .

THEOREM 5.3. The MCP condition. Given a list of trajectories $[T_1, T_2, \dots, T_p]$ sorted by their start times in an increasing order (i.e., $T_i.startTime \leq T_{i+1}.startTime$), a necessary but not sufficient condition for these trajectories to compose a joinable subset is that the location sequence for the MCP of any $\{T_1, T_2, \dots, T_{p-k}\} (0 \leq k < p)$ must be a prefix of a valid path in \mathcal{G}_t .

According to Theorem 5.3, when generating cliques, if the vertices are added to C in an increasing order of the start times of their corresponding trajectories, we can prune some unnecessary vertex combinations according to the MCP condition. The checking is performed with a *pck* predicate.

The *pck* predicate. Similar to the *cex* and *jnb* predicates, given a trajectory graph \mathcal{G}_t , the *pck* predicate can be applied on one or more trajectories and evaluates to *true* iff the MCP condition holds, i.e., $\{\mathcal{T}_x | (pck(\mathcal{T}_x))\} = \{\mathcal{T}_x | \exists P(r_1.loc \rightarrow \dots \rightarrow r_k.loc = prefix(P)), [r_1, \dots, r_k]$ is the MCP of \mathcal{T}_x and $prefix(P)$ is the prefix of a valid path P in \mathcal{G}_t . In terms of “restrictiveness”, this predicate falls somewhere between the *cex* and *jnb* predicates. Compared with *cex*, it further requires that the location sequence must be a prefix of a valid path, not just a subsequence;

compared with *jnb*, it just ensures that the first location is an entrance location. Due to space limitation, the detailed algorithm for evaluating this predicate is omitted.

With the *pck* predicate defined, we try to modify the qualified-clique generation algorithm by pruning worthless results and recursions. First of all, we must ensure that the cliques are generated in the order of their trajectories’ start times. Fortunately, Algorithm 2 iterates through the vertices with an index list L . Thus, to keep the generation order, we just need to sort the vertices in \mathcal{G}_m . Then, each time before outputting a generated clique to the result set, we check its corresponding trajectory set with the *pck* predicate, and only if it evaluates to *true*, can we accept the clique and continue adding more vertices into the result set. The modified algorithm snippet is shown in Algorithm 4.

Algorithm 4 Clique generation with pruning

```

...
sort vertices in  $\mathcal{G}_m$  by their corresponding trajectories’ start
times in descending order
function CLIQUE( $C, L$ )
...
if pck( $C.trajectories$ ) = true then
    results  $\leftarrow$  results  $\cup$   $\{C\}$ 
    if  $\neg L_{new}.empty()$  and  $C.RecordNumber < \theta$  and
 $|C| < \zeta$  then
        CLIQUE( $C, L_{new}$ )
...

```

Example 5.4. Suppose that the vertices in Figure 5(a) are already sorted by the start times of their corresponding trajectories, i.e., $T_5.startTime \leq T_4.startTime \leq \dots \leq T_1.startTime$. If the MCP condition does not hold on $\{T_5\}$, any cliques containing v_5 (e.g., $\{v_5, v_4\}$ and $\{v_5, v_4, v_1\}$) will be pruned by the modified algorithm. For the same reason, if the MCP condition does not hold on $\{T_5, T_2\}$, the cliques $\{v_5, v_2\}$ and $\{v_5, v_2, v_1\}$ will be pruned. Obviously, the modified algorithm is more efficient thanks to the pruning of some cliques and unnecessary calculations.

6 EXPERIMENTS

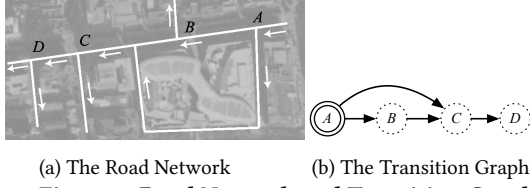
We conduct extensive experiments on both real and synthetic datasets to thoroughly evaluate the properties of the proposed approach and compare it to baseline methods.

6.1 Experimental Settings

All algorithms are implemented in Java and run on a desktop PC with a 2.5GHz Intel i5 CPU and 8GB of memory. Each set of experiments are repeated at least 30 times and the average results are recorded.

A real dataset and a series of synthetic datasets with different characteristics are used in the experiments. According to Section 2.3, the repair approach we proposed in this paper is in the interest of local regions. For such small regions, the transition graphs may seem simple. However, note that even for such seemingly simple graphs, the repair problem is still quite challenging, as revealed in Section 3.

6.1.1 Datasets. Real Dataset. The real dataset is obtained from a real traffic surveillance system in a provincial capital in China. We choose a specific region of this city and extract 699 trajectories of vehicles which contain 2,045 tracking records between 8:00 a.m. and 9:00 a.m. on a particular day. Figure 9(a) illustrates the road network and the distribution of surveillance



(a) The Road Network (b) The Transition Graph

Figure 9: Road Network and Transition Graph

cameras in this region. The license plate numbers of the vehicles are captured by cameras located at A , B , C , and D whenever they pass by these sites. Figure 9(b) is the corresponding transition graph we derived. Due to OCR errors and other issues, some of the license plates in the dataset were misidentified. We manually label the plate numbers by examining the *original photos* taken by the cameras, which serves as the ground truth. In this way, we obtain a labeled dataset that contains both the raw and the true values. The default values of θ , η , ζ and λ for the real dataset are empirically set to 4, 600 seconds, 4, and 0.5, respectively, unless otherwise specified.

Synthetic Datasets. To generate a synthetic trajectory set for ID repair, we first choose a transition graph, based on either the real dataset or a sample of the California road network [21]. Then we repeatedly sample random valid paths and generate corresponding trajectories until we have obtained the desired number of trajectories. Without loss of generality, we assume that an ID consists of 7 to 9 lower-case letters only, which are independently and identically generated following a uniform distribution. The time span is sampled from the empirical distribution of travel time between the corresponding locations in the real dataset. After that, using the edit distance distribution for erroneous IDs in the real dataset as a ballpark, we randomly inject ID errors to the tracking records with a specified error rate, and eventually get a synthetic dataset. The default error rate is set to 20%, unless otherwise specified.

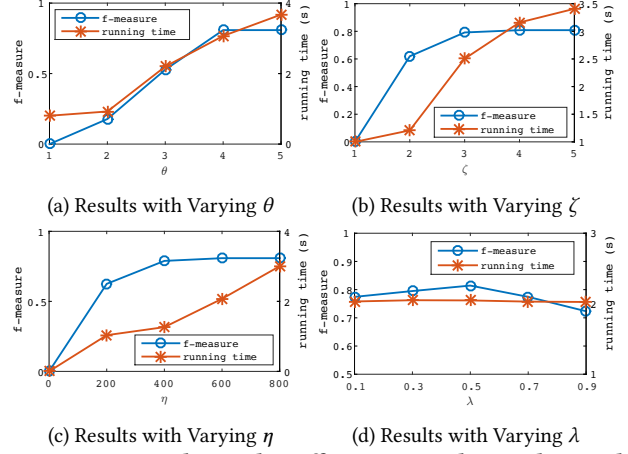
6.1.2 Metrics. We use elapsed time as the metric for efficiency, and adopt *precision*, *recall* and *f-measure* as the general metrics for effectiveness. Using \mathcal{T}_e to represent all the trajectories with ID errors, \mathcal{T}_r to represent those trajectories with ID rewritten by applying candidate repairs, and \mathcal{T}_c to represent all the trajectories whose IDs are correctly repaired, we define $\text{recall} = |\mathcal{T}_c|/|\mathcal{T}_e|$, $\text{precision} = |\mathcal{T}_c|/|\mathcal{T}_r|$, and $\text{f-measure} = \frac{2 \cdot (\text{precision} \cdot \text{recall})}{(\text{precision} + \text{recall})}$. There are also some specialized metrics used in certain groups of experiments, which will be introduced later.

6.2 Effects of Parameters

We first evaluate the effects of different parameters through a group of experiments on the real dataset.

The effects of θ , ζ , and η . Figures 10(a), 10(b) and 10(c) show the f-measure and running time with varying values of θ , ζ , and η , respectively, with all other parameter values fixed at their default values. We observe that for each of these parameters, the running time grows with increasing parameter values. For the f-measure, it initially increases as well, but eventually flattens out. This verifies our earlier observation that for a particular dataset, there exist bounds on these parameters, beyond which no further gains in repair effectiveness can be achieved. Thus, by carefully choosing the bounds, we can reduce the running time of the repair process significantly.

The effect of λ . Figure 10(d) shows the effect of λ in Equation (3). With λ varying from 0.1 to 0.9, the running time remains stable, and the f-measure first increases and then decreases after



(c) Results with Varying η (d) Results with Varying λ

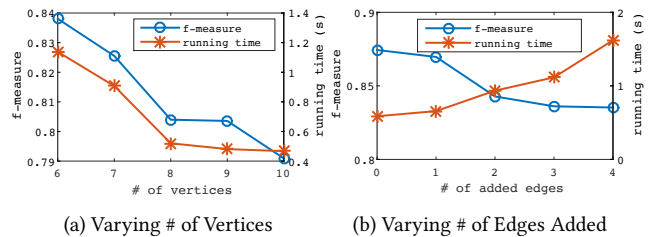
Figure 10: Results with Different Bounds on the Real Dataset

$\lambda = 0.5$. The results imply that (1) there exists an optimal λ value with which the best results can be produced, and (2) the repair results are not sensitive to changes in λ .

6.3 Effects of Data Characteristics

We next conduct a set of experiments using synthetic datasets to investigate the impact of different data characteristics. All the datasets used in this set of experiments are produced based on 500 original trajectories (before injecting errors). The actual number of trajectories in a dataset is affected by different parameters (e.g., error rate and record missing rate), and will be shown for different groups of experiments. The default values of θ , η , ζ and λ for the synthetic datasets used here are set to 8, 600 seconds, 4, and 0.5, respectively.

6.3.1 Size and Density of the Transition Graph. The first data characteristics we explore are the size (number of vertices) and density (number of edges) of the transition graph. The experiments are conducted on synthetic trajectory sets generated from transition graphs with different sizes and different densities. We vary the density of a transition graph with 8 vertices $\mathcal{G}_t = (\mathbf{V}, \mathbf{E}, \mathbf{I}, \mathbf{O})$, where $\mathbf{V} = \{loc_1, loc_2, \dots, loc_8\}$, $\mathbf{E} = \{(loc_1, loc_2), (loc_2, loc_3), \dots, (loc_7, loc_8)\}$, $\mathbf{I} = \{loc_1\}$, and $\mathbf{O} = \{loc_8\}$, by randomly adding a specific number of edges (without duplicate) to it.



(a) Varying # of Vertices (b) Varying # of Edges Added

Figure 11: Effect of the Size and Density of Transition Graphs

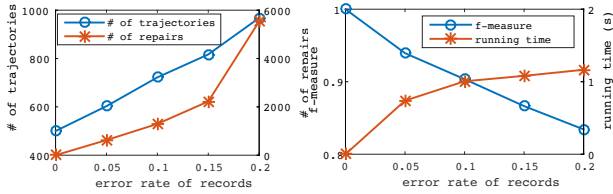
The effect of graph size. Figure 11(a) shows the results on varying transition graph sizes. It is evident that both the f-measure and the running time decrease with the number of vertices increasing. That is because a transition graph with more vertices tends to have longer valid paths, and the longer the valid paths are, the harder it is to form candidate repairs that could "reassemble" the original trajectory;

The effect of graph density. The results for adding varying number of edges to a given transition graph are shown in 11(b).

The f-measure decreases while the running time increases with more edges added, due to the following reasons: (1) adding edges to the transition graph will increase the number of valid paths and thus there will be more candidate repairs; (2) with the number of candidate repairs growing, there may be more false positive repairs (vertices) being selected and that will cause the f-measure to deteriorate; and (3) having more candidate repairs also leads to longer candidate generation and selection time, resulting in an increase in the total running time.

The results above imply that our ID repair approach is more suitable for sparse transition graphs with limited number of vertices, which is actually the case in many, if not most, application scenarios. This is also consistent with our assumptions and analysis made in Section 2.3.

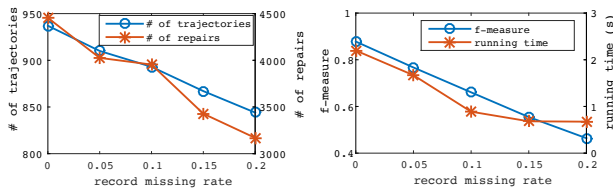
6.3.2 ID Error Rate. To evaluate the effect of the ID error rate, we create a cohort of synthetic datasets by randomly injecting ID errors, each time with a different error rate, into an identical original trajectory set.



(a) # of Trajectories/Repairs (b) F-Measure and Running Time
Figure 12: Effect of ID Error Rate

The experiment results are reported in Figures 12(a) and 12(b), from which we can observe that with the error rate increasing, (1) the number of trajectories for the input dataset increases linearly; (2) both the number of candidate repairs and the running time increase polynomially; and (3) the f-measure drops near linearly. The reason is as follows. Since ID errors can cause a trajectory to break into multiple pieces, the input number of trajectories grows linearly with respect to the error rate. Both the number of candidate repairs and the running time also increase accordingly. The f-measure drops mainly because intuitively it is more difficult to “reassemble” the original trajectory with more IDs misidentified. Also, recall that our repair approach assumes that all the correct IDs exist in the dataset, which may no longer hold if the error rate gets high. In summary, the lower the ID error rate is, the better our repair approach works.

6.3.3 Record Missing Rate. As mentioned in Section 1, in this work we only consider errors caused by ID misidentification, ignoring the effect of missing records. In practice, however, there may be a slight chance of record missing from the dataset. We thus conduct experiments to evaluate whether this has a significant impact on the effectiveness of the proposed approach. To this end, we first generate a synthetic dataset and then randomly remove records from it with varying record missing rates.



(a) # of Trajectories/Repairs (b) F-Measure and Running Time
Figure 13: Effect of Record Missing

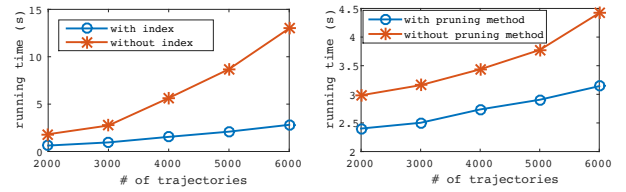
As illustrated in Figures 13(a) and 13(b), with the missing rate increasing from 0% to 20%, all the metrics decrease. The reason is that (1) record missing will make some joinable subsets incomplete and thus cannot compose the corresponding candidate repairs (this is verified by the decrease of candidate repairs shown in Figure 13(a)); (2) trajectories belonging to different entities may be joined due to the absence of some trajectories; and (3) records containing the true ID for an entity may have all been removed, which makes some errors irreparable.

According to the experiment result, although having missing records has a notable impact on the effectiveness of the proposed ID repair approach, it is still applicable for datasets with relatively low record missing rates.

6.4 Effectiveness of the Optimization Methods

The main purpose of the next group of experiments is to explore the performance improvements brought by the Length-Indexed Grids (in Section 5.1), as well as the pruning method (in Section 5.2).

We conduct the experiments on synthetic datasets with the number of trajectories varying from 2,000 to 6,000 and the corresponding number of records varying from 5,189 to 15,795. All the datasets are generated using the same transition graph as that for the real dataset.



(a) Construction Time for G_m (b) Running Time with Different Data Sizes
Figure 14: Effectiveness of Optimization Methods

Figure 14(a) shows the running time of the trajectory graph construction process with different number of trajectories. From this figure we can observe that without indexing, the construction time of G_m grows superlinearly with the number of trajectories, whereas the trend becomes almost linear with the Length-Indexed Grids. This observation indicates that the Length-Indexed Grids can help eliminate a large number of unnecessary trajectory comparisons.

Figure 14(b) reports the running time of the whole repair process with the number of trajectories varying from 2,000 to 6,000. We can see that the time increases polynomially with the number of trajectories. Besides, compared with the basic clique generation algorithm, algorithm with the pruning method can reduce about 30% running time.

6.5 Comparison with Competing Approaches

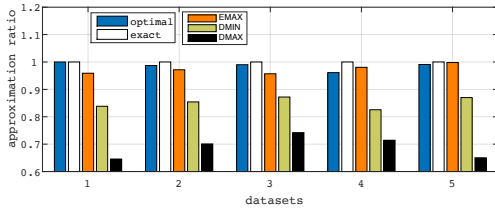
To evaluate the effectiveness of our proposed method, we compare it with other approaches that use different repair selection algorithms or exploit different constraints.

6.5.1 Alternative Repair Selection Algorithms. In this set of experiments, we aim to investigate the performance of different algorithms for the repair selection phase. In addition to EMAX and the exact algorithms introduced in Section 4.2, we also implement three other algorithms for comparison. The first algorithm,

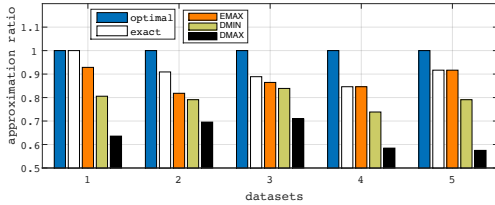
named *optimal selection*, is an oracle machine based algorithm that always selects and applies correct candidate repairs regardless of their ω values. Theoretically, this algorithm can achieve the highest quality improvement. The second and the third algorithms are minimum degree first (DMIN) and maximum degree first (DMAX). As their names suggest, they select the vertex with the minimum/maximum degree from \mathcal{G}_r in each step and discard adjacent vertices until there is no vertex left.

As the exact algorithm for weighted-independent set is time consuming, the experiments are conducted on 5 small synthetic datasets whose sizes do not exceed 100. Even so, the average running time for the exact algorithm is still thousands times longer than the other algorithms. Thus we only report on their effectiveness rather than the performance.

To measure the real quality of a dataset, we employ the metric *trajectory accuracy*, which is defined as the ratio of trajectories with correct IDs. Thereby, the real quality improvement after repairing can be measured by the increment in this metric. As trajectory merging can change the data size, we will only perform ID rewritings. Using ΔE (ΔA) and ΔE_{max} (ΔA_{opt}) to represent the selected Ω value (trajectory accuracy improvement) and the maximum selected Ω value (maximum trajectory accuracy improvement), the approximation ratio for maximum Ω value selection and data quality improvement can be calculated by $\Delta E/\Delta E_{max}$ and $\Delta A/\Delta A_{opt}$.



(a) Approximation Ratios for Repairs Selection



(b) Approximation Ratios for Quality Improvement

Figure 15: Approximation Ratios for Different Selection Algorithms on Synthetic Datasets

Figures 15(a) and 15(b) report the experiment results of maximum Ω value selection and data quality improvement, from which we can observe that (1) the selected Ω value can reflect the data quality improvement well, (2) the total selected Ω value for the optimal selection algorithm is randomly distributed around, rather than always coincides with, the maximum value, and (3) remarkably, the proposed EMAX algorithm can achieve an average approximation ratio of more than 0.95 and 0.85 for repair selection and real data quality improvement respectively, which significantly beats the other two heuristic algorithms. In summary, as the optimal selection algorithm is evasive in practice and the exact algorithm is time-consuming, the proposed EMAX algorithm seems highly promising.

6.5.2 Comparison with Other Repair Approaches. To evaluate our proposed ID repair approach, we implement a baseline approach based on ID similarity = 3, i.e., trajectories with ID

similarity ≤ 3 are considered to come from the same entity and thus will be merged. Also, we implement another greedy heuristic method based on neighborhood constraints proposed in [27]. We take the transition graph \mathcal{G}_t as the constraint graph and the trajectory graph \mathcal{G}_m as the instance graph. The cost function is set to be the edit distance of two ID strings. To make sure the algorithm terminates, we add a variation to the approach that edges are allowed to be removed from \mathcal{G}_m during relabeling.

The repair results of the three approaches are shown in Figure 16, from which we can observe that (1) while the precision of the other competing approaches is somewhat close to our proposed approach, their recall is significantly lower; and (2) the neighborhood constraint based method performs even worse than the baseline method for our problem. The recall of the ID similarity based approach is better than that of the neighborhood constraint based approach because it supports “partial recovery” of the original trajectories. Actually, both the ID similarity based approach and the neighborhood constraint based approach are binary constraints that only consider the relationship between trajectories pairs. In contrast, our transition graph based approach considers the relationships between multiple trajectories, which is why it can cover more correct repairs.

7 RELATED WORK

There has been a sizable body of work in the areas of data repair and data matching that can be considered related to our work, which we summarize below.

7.1 Data Repair

Most previous work on data repair has focused on relational data by exploiting the different types of dependencies [1], e.g., matching dependencies [12], differential dependencies [26], and order dependencies [28]. Fan et al. extend the functional and inclusion dependencies with conditions [6, 13] and also extend their data inconsistency detection method to distributed environments [14]. Although highly successful, most of the work has not considered spatial and temporal factors.

Moreover, sequential dependencies [18] are developed to constrain attributes’ transitions. Song et al. use neighborhood constraints to repair vertex labels in graphs [27]. Wang et al. employ the Petri Net to repair the names of event logs [29]. Similar to our study that focuses on repairing the IDs, Song et al. propose a method for cleaning timestamps facilitated by temporal constraints [25].

For unified approaches, Ilyas et al. propose a novel holistic repairing algorithm [8], as well as a general system [11] that puts multiple constraints into consideration and repairs them all at once. Similarly, Geerts et al. develop a uniform data-cleaning framework with a cell group and partial order based cleaning semantics [17]. However, those methods cannot be trivially adopted, since it is difficult to transform the constraints posed by transition graphs in our setting into the denial constraints or equality-generating dependencies required by those methods.

7.2 Data Matching

In the field of data matching, Yakout et al. try to identify the same entity in different transactions by detecting regularity patterns from merged behavior logs [32]. Similarly, Zhu et al. perform heterogeneous event matching by finding an optimal mapping that can maximize the frequency similarity of patterns [33].

When patterns are not explicitly given, Li et al. propose a temporal model, as well as an algorithm to perform temporal

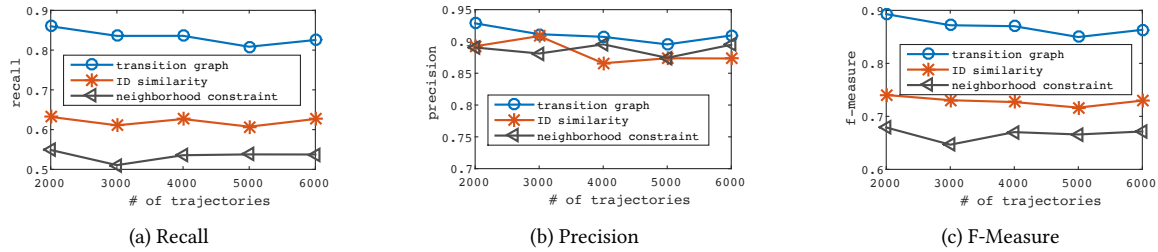


Figure 16: Comparisons with Other Repair Approaches

records clustering [22]. They utilize both the usual similarity metrics and the temporal model with collected evidences to make the decision. Chiang et al. extend their work and develop a two-phase method called “static first, dynamic second” to reduce the complexity of the temporal model [7]. Also, they use signatures to improve the computing efficiency. Note that both their work and ours are to identify entities by exploring their transitions. The main difference is that while their work mainly focuses on when the state attributes of entities should change, we focus on how (through which paths) the entities pass through the area of interest.

8 CONCLUSIONS AND FUTURE WORK

We have studied a novel problem of repairing erroneous IDs in spatio-temporal trajectories with transition graphs. A two-phase repair paradigm, which includes candidate repair generation and compatible repair selection, is proposed to address this problem. Since both phases are intractable in general, we exploit the locality and sparsity properties of trajectories and present efficient solutions in restricted but practical scenarios. For the candidate repair generation phase, we propose a backtracking algorithm, as well as a pruning method to speed it up. For the candidate repair selection phase, we present a practical greedy algorithm. Extensive experiments are conducted on both real and synthetic data to study the effects of various parameters and data characteristics. In addition, we compare our proposed approach with some baseline methods and the results have confirmed its effectiveness.

One possible direction for future work would be to deploy our algorithms on some distributed repair systems with UDF support [20]. It would also be interesting to study solutions that could perform ID repair as the tracking records stream in.

ACKNOWLEDGMENTS

This work was supported in part by the National Basic Research 973 Program of China under Grant No. 2015CB352502, the National Natural Science Foundation of China under Grant Nos. 61272092 and 61572289, the Natural Science Foundation of Shandong Province of China under Grant No ZR2015FM002, and the NSERC Discovery Grants.

REFERENCES

- [1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of databases*. Vol. 8.
- [2] S Sheik Mohammed Ali, Bobby George, Lelitha Vanajakshi, and Jayashankar Venkatraman. 2012. A multiple inductive loop vehicle detection system for heterogeneous and lane-less traffic. *IEEE Transactions on Instrumentation and Measurement* 61, 5 (2012), 1353–1360.
- [3] Jorge A Alves. 2001. *Recognition of ship types from an infrared image using moment invariants and neural networks*. Technical Report. NAVAL POST-GRADUATE SCHOOL MONTEREY CA.
- [4] Philip Bohannon, Michael Flaster, Wenfei Fan, and Rajeev Rastogi. 2005. A Cost-Based Model and Effective Heuristic for Repairing Constraints by Value Modification. In *SIGMOD*. 143–154.
- [5] Immanuel M Bomze, Marco Budinich, Panos M Pardalos, and Marcello Pelillo. 1999. The maximum clique problem. In *Handbook of combinatorial optimization*. 1–74.
- [6] Loreto Bravo, Wenfei Fan, and Shuai Ma. 2007. Extending Dependencies with Conditions. In *Vldb*. 243–254.
- [7] Yueh-Hsuan Chiang, AnHai Doan, and Jeffrey F. Naughton. 2014. Tracking Entities in the Dynamic World: A Fast Algorithm for Matching Temporal Records. *PVLDB* 7, 6 (2014), 469–480. <http://www.vldb.org/pvldb/vol7/p469-chiang.pdf>
- [8] Xu Chu, Ihab F. Ilyas, and Paolo Papotti. 2013. Holistic data cleaning: Putting violations into context. In *ICDE*. 458–469.
- [9] Dong Deng, Guoliang Li, Jianhua Feng, and Wen-Syan Li. 2013. Top-k string similarity search with edit-distance constraints. In *ICDE*. 925–936.
- [10] Shan Du, Mahmoud Ibrahim, Mohamed Shehata, and Wael Badawy. 2013. Automatic license plate recognition (ALPR): A state-of-the-art review. *IEEE Transactions on circuits and systems for video technology* 23, 2 (2013), 311–325.
- [11] Amr Ebaid, Ahmed Elmagarmid, Ihab F Ilyas, Mourad Ouzzani, Jorge-Arnulfo Quiane-Ruiz, Nan Tang, and Si Yin. 2013. NADEEF: A generalized data cleaning system. *PVLDB* 6, 12 (2013), 1218–1221.
- [12] Wenfei Fan. 2008. Dependencies revisited for improving data quality. In *PODS*. 159–170.
- [13] Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. 2008. Conditional functional dependencies for capturing data inconsistencies. *ACM Trans. Database Syst.* 33 (2008).
- [14] Wenfei Fan, Floris Geerts, Shuai Ma, and Heiko Müller. 2010. Detecting inconsistencies in distributed data. In *ICDE*. 64–75.
- [15] Joao C Ferreira, Jorge Branquinho, Paulo Chaves Ferreira, and Fernando Piedade. 2017. Computer Vision Algorithms Fishing Vessel Monitoring—ATI-identification of Vessel Plate Number. In *International Symposium on Ambient Intelligence*. Springer, 9–17.
- [16] Robert W Floyd. 1962. Algorithm 97: shortest path. *Commun. ACM* 5, 6 (1962), 345.
- [17] Floris Geerts, Giansalvatore Mecca, Paolo Papotti, and Donatello Santoro. 2013. The LLUNATIC data-cleaning framework. *PVLDB* 6, 9 (2013), 625–636.
- [18] Lukasz Golab, Howard J. Karloff, Flip Korn, Avishek Saha, and Divesh Srivastava. 2009. Sequential Dependencies. *PVLDB* 2, 1 (2009), 574–585.
- [19] Richard M. Karp. 1972. Reducibility Among Combinatorial Problems. In *Proceedings of a Symposium on the Complexity of Computer Computations*. 85–103. <http://www.cs.berkeley.edu/~luca/cs172/karp.pdf>
- [20] Zuhair Khayyat, Ihab F Ilyas, Alekh Jindal, Samuel Madden, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Si Yin. 2015. Bigdancing: A system for big data cleansing. In *SIGMOD*. ACM, 1215–1230.
- [21] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. <http://snap.stanford.edu/data>. (June 2014).
- [22] Pei Li, Xin Luna Dong, Andrea Maurino, and Divesh Srivastava. 2011. Linking Temporal Records. *PVLDB* 4, 11 (2011), 956–967.
- [23] R Garey Michael and S Johnson David. 1979. Computers and intractability: a guide to the theory of NP-completeness. *WH Free. Co., San Fr* (1979), 90–91.
- [24] Sunita Sarawagi and Alok Kirpal. 2004. Efficient set joins on similarity predicates. In *SIGMOD*. 743–754.
- [25] Shaoxu Song, Yue Cao, and Jianmin Wang. 2016. Cleaning timestamps with temporal constraints. *PVLDB* 9, 10 (2016), 708–719.
- [26] Shaoxu Song and Lei Chen. 2011. Differential dependencies: Reasoning and discovery. *ACM Trans. Database Syst.* 36, 3 (2011), 16.
- [27] Shaoxu Song, Hong Cheng, Jeffrey Xu Yu, and Lei Chen. 2014. Repairing Vertex Labels under Neighborhood Constraints. *PVLDB* 7, 11 (2014), 987–998.
- [28] Jaroslav Szlichta, Parke Godfrey, Jarek Gryz, and Calisto Zuzarte. 2013. Expressiveness and Complexity of Order Dependencies. *PVLDB* 6, 14 (2013), 1858–1869.
- [29] Jianmin Wang, Shaoxu Song, Xuemin Lin, Xiaochen Zhu, and Jian Pei. 2015. Cleaning structured event logs: A graph repair approach. In *ICDE*. 30–41.
- [30] Jianmin Wang, Shaoxu Song, Xiaochen Zhu, and Xuemin Lin. 2013. Efficient Recovery of Missing Events. *PVLDB* 6, 10 (2013), 841–852.
- [31] Andreas Wombacher. 2011. A-posteriori detection of sensor infrastructure errors in correlated sensor data and business workflows. *Business Process Management* (2011), 329–344.
- [32] Mohamed Yakout, Ahmed K. Elmagarmid, Hazem Elmeleegy, Mourad Ouzzani, and Alan Qi. 2010. Behavior Based Record Linkage. *PVLDB* 3, 1 (2010), 439–448.
- [33] Xiaochen Zhu, Shaoxu Song, Jianmin Wang, Philip S. Yu, and Jianguang Sun. 2014. Matching heterogeneous events with patterns. In *ICDE*. 376–387.

MTBase: Optimizing Cross-Tenant Database Queries

Lucas Braun*
Oracle Labs
lucas.braun@oracle.com

Donald Kossmann*
Microsoft Research
donald@microsoft.com

Renato Marroquín
Systems Group, Department
of Computer Science, ETH Zurich
marenato@inf.ethz.ch

Ken Tsay*
Careem Networks GmbH
ken.tsay@careem.com

ABSTRACT

In the last decade, many business applications have moved into the cloud. In particular, the “database-as-a-service” paradigm has become mainstream. While existing multi-tenant data management systems focus on single-tenant query processing, we believe that it is time to rethink how queries can be processed across multiple tenants in such a way that we do not only gain more valuable insights, but also at minimal cost. As we will argue in this paper, standard SQL semantics are insufficient to process cross-tenant queries in an unambiguous way, which is why existing systems use other, expensive means like ETL or data integration instead. We first propose MTSQL, an extension to standard SQL, which fixes the ambiguity problem. Next, we present MTBase, a query processing middleware that efficiently processes MTSQL on top of SQL. As we will see, there is a canonical, provably correct, rewrite algorithm from MTSQL to SQL, which may however result in poor query execution performance, even on high-performance database products. We further show that with carefully-designed optimizations, execution times can be reduced in such ways that the difference to single-tenant queries becomes marginal.

1 INTRODUCTION

Indisputably, cloud computing is one of the fastest growing businesses related to the field of computer science. Cloud providers promise good elasticity, high availability and a fair pay-as-you-go pricing model to their tenants. Moreover, corporations are no longer required to rely on on-premise infrastructure which is typically costly to acquire and maintain. While it is still an open research question whether and how these good promises can be kept with regard to databases [19, 32], all the big players, like Google [30], Amazon [8], Microsoft [34] and recently Oracle [38], have launched their own Database-as-a-Service (DaaS) cloud products.

All these products host massive amounts of data from multiple clients and are therefore *multi-tenant*. However, as pointed out by Chong et al. [17], the term *multi-tenant database* is ambiguous and can refer to a variety of DaaS schemes with different degrees of logical data sharing between tenants. On the other hand, as argued by Aulbach et al. [11], multi-tenant databases not only differ in the way how tenants logically share information, but also how information is physically separated. We conclude that the *multi-tenancy spectrum* consists of four different schemes: First, there are DaaS products that offer each tenant her proper database while relying on shared resources (*SR*), i.e. hardware (e.g.

CPU, network, storage) and/or software (e.g. buffer pools, system tables, system users, etc.). Examples include *SAP HANA* [42], *SqlVM* [36], *RelationalCloud* [35], *Snowflake* [18] and *Oracle’s multitenant container database (CDB)* [40]. Next, there are systems that share databases (*SD*), but each tenant gets her own set of tables within such a database, as for instance *Azure SQL DB* [20].

Finally, there are the two schemes where tenants not only share a database, but also the table layout (schema). Either, as for example in *Apache Phoenix* [9], tenants still have their private tables, but these tables share the same (logical) schema (*SS*), or the data of different tenants is consolidated into shared tables (*ST*) which is hence the layout with the highest degree of physical and logical sharing. Prominent examples for *ST* include Oracle’s Virtual Private Database [3] as well as different Microsoft Azure DaaS offerings [33, 34]. *SS* and *ST* layouts are not only used in DaaS, but also in Software-as-a-Service (SaaS) platforms, as for example in *Salesforce* [44]. The main reason why all these commercial systems prefer *ST* over *SS* is cost [11]. Moreover, if the number of tenants exceeds the number of tables a database can hold, which is typically a number in the range of ten thousands, *SS* becomes prohibitive. Conversely, *ST* databases can easily accommodate hundred thousands to even millions of tenants.

An important feature of *multi-tenant databases*, which, to the best of our knowledge, no DaaS or SaaS natively supports today, is *cross-tenant query processing*, i.e. combining data of different tenants and query this unified data set as if it was single-tenant, using SQL. In order to illustrate that *cross-tenant query processing* is indeed a highly relevant requirement, let us have a look at one of the many initiatives to democratize the use of personal data, the *Health Data Cooperative (HDC)* [27]. In HDC, all patient data is stored in a single, multi-tenant SaaS database, each patient being a tenant managing her own data. For clinical studies, however, it is essential to be able to run queries over a cohort of patients who give their consent, or, in other words enable *cross-tenant query processing*. Clearly, the health data use case has also another big challenge, which is data privacy. This aspect, despite being out of the scope of this paper, is considered essential future work.

There are several existing approaches to *cross-tenant query processing* which are summarized in Figure 1. The first approach is *data warehousing* [29] where data is *extracted* from several *data sources* (tenant databases/tables), *transformed* into one common format and finally *loaded* into a new database where it can be queried by the client. This approach has high integration transparency in the sense that once the data is loaded, it is in the expected format as required by the client and she can ask any query she wants, using plain SQL. Moreover, as all data is in a single place, queries can be optimized. On the down-side of this approach – well-known and argued by many [10, 14, 37] – are costs in terms of both, developing and maintaining such *ETL*

*most of the work performed while at ETH

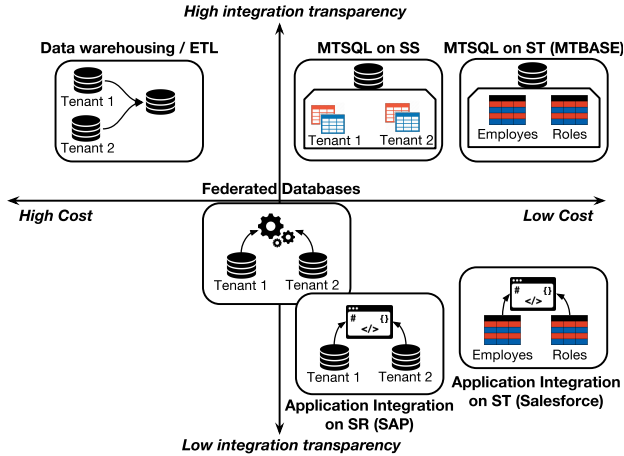


Figure 1: Cross-tenant query processing systems

pipelines, as well as maintaining a separate copy of the data. Another disadvantage is *data staleness* in the presence of frequent updates.

Federated Databases [26, 31] reduce some of these costs by integrating data *on demand*, i.e. there is no data copying. However, maintenance costs are still significant as for every new data source, a new integrator/wrapper has to be developed. As data resides in different places (and different formats), queries can only be optimized to a very small extent (if at all), which is why the degree of integration transparency is considered sub-optimal. Finally, systems like *SAP HANA* [42] and *Salesforce* [44], which are mainly tailored towards single-tenant queries, offer some degree of *cross-tenant query processing*, but only through their application logic, not natively. This means that the set of queries that can be asked is limited, accounting for low integration transparency.

We believe that the reason why none of these previous works uses a native approach, i.e. SQL plus transparent rewriting, for *cross-tenant query processing* is that there is an *ambiguity problem*.¹ Consider, for instance, the *ST* database in Figure 2, which we are going to use as a running example throughout the paper. Further assume that we would like to query the joint dataset of tenants 0 and 1: As shown on the left, we might want to join *Employees* with *Roles*. Joining on *role_id* alone is not enough as this would also join *Alice* with *executive*, which does not correspond to the expected output because *Alice* is a professor, and only a professor. In this case, a rewrite algorithm would have to add the tenant-ID *ttid* to the join predicate. On the other hand, joining the *Employees* table with itself on *E1.age = E2.age*, as illustrated on the right, does not require *ttid* to be present in the join predicate because it actually makes sense to include results like (*Alice*, *Ed*) because they are indeed the same age.

An additional challenging fact is that different tenants might store their data in different units. In our example, tenant 0 might store her employees’ salaries in a different currency than tenant 1. If this is the case, computing the average salary across all tenants clearly involves some value conversions that should, ideally, happen without the client noticing or even worrying about.

This paper presents *MTSQL* as a solution to these ambiguity problems, following a native approach. *MTSQL* extends the SQL API and provides additional data definition syntax and corresponding semantics specifically-suited for *cross-tenant query*

¹Note, however, that SQL plus transparent rewriting works for *single-tenant query processing* in a multi-tenant system. Apache Phoenix [9] and Oracle’s Virtual Private Database [3] do exactly that.

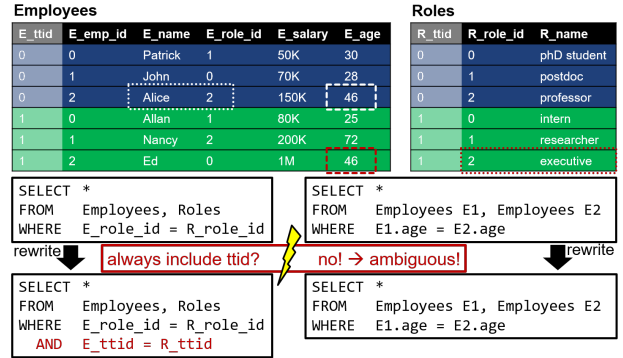


Figure 2: Multi-tenant database in *basic layout (ST)*, illustrating the ambiguity problem in *cross-tenant queries*

processing. It enables high integration transparency because once the schema is defined and the database connection established, any client, with any desired data format, can ask any query at any time and do so by using nothing else but plain SQL. Moreover, as data resides in a single database (*SS* or *ST*), queries can be aggressively optimized with respect to both, standard SQL semantics and additional *MTSQL* semantics. As *MTSQL* adopts the single-database layout, it is also very cost-effective, especially if used on top of *ST*. Also, data conversion only happens as needed, which perfectly fits the cloud’s *pay-as-you-go* cost model and thus makes *MTSQL* an attractive option to complement existing *DaaS* offerings. Specifically, the paper makes the following contributions:

- It defines the syntax and semantics of *MTSQL*, a database language that extends SQL and solves the ambiguity problem for *cross-tenant query processing*.
- It presents the design and implementation of *MTBase*, a database middleware that executes *MTSQL* on top of any *shared-table* multi-tenant database.
- It studies *MTSQL*-specific optimizations for query execution in *MTBase*.
- It extends the well-known TPC-H benchmark in order to run and evaluate *MTSQL* workloads, resulting in new benchmark called *MT-H*.
- It evaluates the performance and the implementation correctness of *MTBase* with *MT-H*, concluding with satisfactory results.

The rest of this paper is organized as follows: Section 2 defines *MTSQL*, while Section 3 gives an overview on *MTBase*. Section 4 discusses the *MTSQL*-specific optimizations which are validated in Section 5. Section 6 shortly summarizes lines of related work, specifically focusing on the relation of *MTSQL* to data integration as well as data privacy, whereas the paper is concluded in Section 7.

2 MTSQL

In order to model the specific aspects of *cross-tenant query processing* in *multi-tenant databases*, we developed *MTSQL*, which will be described in this section. *MTSQL* extends SQL in two ways: First, it extends the SQL interface with two additional parameters, *C* and *D*. *C* is the tenant ID (or *ttid* for short) of the client who submits a statement and hence determines the format in which the result must be presented. The data set, *D*, is a set of *ttids* that refer to the tenants whose data the client wants to query. Secondly, *MTSQL* extends the syntax and semantics of SQL, as well as its Data Definition Language (DDL), Data Manipulation

Language (DML) and Data Control Language (DCL, consists of GRANT and REVOKE statements).

As mentioned in the introduction, there are several ways how a multi-tenant database can be laid out: Figure 2 shows an example of the *ST* scheme, also referred to as *basic layout* in related work [11] where tenants’s data is consolidated using the same tables. Meanwhile, there also exists the *SS* scheme, also referred to as *private table layout*, where every tenant has her own set of tables. In that scheme, *data ownership* is defines as part of the table name (e.g. *Roles_1*, *Roles_2*, ...) while in *ST*, records are explicitly annotated with the *ttid* of their *data owner*, using an extra meta column in the table which is invisible to the client.

As these two approaches are semantically equivalent, the MTSQL semantics that we are about to define, apply to both. In the case of the *SS*, applying a statement *s* with respect to *D* simply means to apply *s* to the logical union of all private tables owned by a tenant in *D*. In *SS*, *s* is applied to tables filtered according to *D*. In order to keep the presentation simple, the rest of this paper assumes an *ST* scheme, but sometimes defines semantics with respect to *SS* if that makes the presentation easier to understand.

2.1 MTSQL API

MTSQL needs a way to incorporate the additional parameters *C* and *D*. As *C* is the *ttid* of the tenant that issues a statement, we assume it is implicitly given by the SQL connection string. *ttids* are not only used for identification and access control, but also for data ownership. While this paper uses integers for simplicity reasons, *ttids* can have any data type, in particular they can also be database user names.

```
SET SCOPE = "IN (1,3,42)";
```

Listing 1: Simple SCOPE expression using IN

```
SET SCOPE = "FROM Employees WHERE E_salary > 180K";
```

Listing 2: Complex SCOPE expression with sub-query

D is defined using the MTSQL-specific SCOPE runtime parameter on the SQL connection. This parameter can be set in two different ways: Either, as shown in Listing 1, as *simple scope* with an IN list stating the set of *ttids* that should be queried, or as in Listing 2, as a sub-query with a FROM and a WHERE clause (*complex scope*). The semantics of the latter is that every tenant that owns at least one record in one of the tables mentioned in the FROM clause that satisfies the WHERE clause is part of *D*. The SCOPE variable defaults to {*C*}, which means that by default a client processes only her own data. Defining a simple scope with an empty IN list, on the other hand, makes *D* include all the tenants present in the database.

Making *C* and *D* part of the connection allowed for a clear separation between the end users of MTSQL (for which *ttids* do not make much sense and hence remain invisible) and administrators/programmers that manage connections (and are aware of *ttids*).

2.2 Data Definition Language

DDL statements are issued by a special role called the *data modeller*. In a multi-tenant application, this would be the SaaS provider (e.g. a Salesforce administrator) or the provider of a specific application. However, the data modeller can delegate this privilege to any tenant she trusts using a GRANT statement, as will be described in Section 2.3.

There are two types of tables in MTSQL: tables that contain common knowledge shared by everybody (like the *Regions* table in *TPC-H* [43]) and those that contain data of a specific tenant (i.e. *Employees* and *Roles* in Figure 2). More formally, we define the *table generality* of *Regions* as *global* and the one of *Employees* as *tenant-specific*. In order to process queries across tenants, MTSQL needs a way to distinguish whether an attribute is *comparable* (can be directly compared against attribute values of other tenants), *convertible* (can be compared against attribute values of other tenants after applying a well-defined *conversion function*) or *tenant-specific* (it does semantically not make sense to compare against attribute values of other tenants). An overview of these types of *attribute comparability*, together with examples from Figure 2, is shown in Table 1.

type	description	examples
comparable	can be directly compared to and aggregated with other values	E_age, R_name
convertible	other values need to be converted to the format of the current tenant before comparison or aggregation	E_salary
tenant-specific	values of different tenants cannot be compared with each other	E_role_id, R.role_id

Table 1: Overview on attribute comparability in MTSQL

2.2.1 CREATE TABLE Statement. The MTSQL-specific keywords for creating (or altering) tables are GLOBAL, SPECIFIC, COMPARABLE and CONVERTIBLE. An example of how they can be used is shown in Listing 3. Note that SPECIFIC can be used for tables and attributes. Moreover, using these keywords is optional as we define that tables are global by default, attributes of tenant-specific tables default to *tenant-specific* and those of global tables to *comparable*.²

```
1 CREATE TABLE Employees SPECIFIC (
2   E_emp_id INTEGER NOT NULL SPECIFIC,
3   E_name VARCHAR(25) NOT NULL COMPARABLE,
4   E_role_id INTEGER NOT NULL SPECIFIC,
5   E_salary VARCHAR(17) NOT NULL CONVERTIBLE
6   @currencyToUniversal @currencyFromUniversal,
7   E_age INTEGER NOT NULL COMPARABLE,
8   CONSTRAINT pk_emp PRIMARY KEY (E_emp_id),
9   CONSTRAINT fk_emp FOREIGN KEY (E_role_id) REFERENCES Roles (
10  R_role_id)
11 );
```

Listing 3: Exemplary MTSQL CREATE TABLE statement, MT-specific keywords marked in bold

2.2.2 Conversion Functions. Cross-tenant query processing requires the ability to execute comparison predicates on *comparable* and *convertible attribute*. While comparable attributes can be directly compared to each other, convertible attributes, as their name indicates, have to be converted first, using conversion functions. Each tenant has a pair of conversion functions for each attribute to translate from and to a well-defined universal format. More formally, a *conversion function pair* is defined as follows:

Definition 2.1. (*toUniversal* : $X \times T \rightarrow X$, *fromUniversal* : $X \times T \rightarrow X$) is a valid MTSQL conversion function pair for attribute *A*, where *T* is the set of tenants in the database and *X* is the domain of *A*, if and only if:

- (i) There exists a *universal format* for attribute *A*:³

$$image(toUniversal(\cdot, t_1)) = image(toUniversal(\cdot, t_2)) = \dots = image(toUniversal(\cdot, t_{|T|}))$$
- (ii) For every tenant $t \in T$, the partial functions $toUniversal(\cdot, t)$ and $fromUniversal(\cdot, t)$ are bijective functions.

²Global tables (shared among all tenants!) can only have comparable attributes anyway.

³ $image(f)$ denotes the mathematical image, i.e. the range of function *f*.

(iii) *fromUniversal* is the inverse of *toUniversal*: $\forall t \in T, x \in X : \text{fromUniversal}(\text{toUniversal}(x, t), t) = x$

These three properties imply the following two corollaries that we are going to need later in this paper:

COROLLARY 1. *toUniversal* and *fromUniversal* are equality preserving: $\forall t \in T : \text{toUniversal}(x, t) = \text{toUniversal}(y, t) \Leftrightarrow x = y \Leftrightarrow \text{fromUniversal}(x, t) = \text{fromUniversal}(y, t)$

COROLLARY 2. Values from any tenant t_i can be converted into the representation of any other tenant t_j by first applying *toUniversal*(\cdot, t_i), followed by *fromUniversal*(\cdot, t_j) while equality is preserved:

$$\forall t_i, t_j \in T : x = y \Leftrightarrow \text{fromUniversal}(\text{toUniversal}(x, t_i), t_j) = \text{fromUniversal}(\text{toUniversal}(y, t_i), t_j)$$

The reason why we opted for a two-step conversion through universal format is that it allows each tenant t_i to define her share of the conversion function pair, i.e. *toUniversal*(\cdot, t_i) and *fromUniversal*(\cdot, t_i), individually without the need of a central authority. Moreover, this design greatly reduces the overall number of partial conversion functions as we need at most $2 \cdot |T|$ partial function definitions, compared to $|T|^2$ functions in the case where we would define a direct conversion for every pair of tenants.

```
1 CREATE FUNCTION phoneToUniversal (VARCHAR(17), INTEGER) RETURNS
  VARCHAR(17)
2 AS 'SELECT SUBSTRING($1, CHAR_LENGTH(PT_prefix)+1) FROM
  Tenant, PhoneTransform WHERE T_tenant_key = $2 AND
  T_phone_prefix_key = PT_phone_prefix_key;'
3 LANGUAGE SQL IMMUTABLE;
```

Listing 4: Converting a phone number to universal form (without prefix), PostgreSQL syntax

```
1 CREATE FUNCTION phoneFromUniversal (VARCHAR(17), INTEGER)
  RETURNS VARCHAR(17)
2 AS 'SELECT CONCAT(PT_prefix, $1) FROM Tenant, PhoneTransform
  WHERE T_tenant_key = $2 AND T_phone_prefix_key =
  PT_phone_prefix_key;'
3 LANGUAGE SQL IMMUTABLE;
```

Listing 5: Converting to a specific phone number format, PostgreSQL syntax

Listings 4 and 5 show an example of such a conversion function pair. These functions are used to convert phone numbers with different prefixes, like “+”, “00” or any other specific county exit code⁴, and the universal format is a phone number without prefix. In this example, converting phone numbers simply means to lookup the tenant’s prefix and then either prepend or remove it, depending whether we convert from or to the universal format. Note that the exemplary code also contains the keyword `IMMUTABLE` to state that for a specific input the function always returns the same output, which is an important hint for the query optimizer. While this keyword is PostgreSQL-specific, some other vendors, but by far not all, offer a similar syntax.

It is important to mention that the *equality-preserving* property as mentioned in Corollary 1 is a minimal requirement for conversion functions to make sense in terms of producing coherent query results among different clients. There are, however conversion functions that exhibit additional properties, for example:

- order-preserving with respect to tenant t :
 $x < y \Leftrightarrow \text{toUniversal}(x, t) < \text{toUniversal}(y, t)$

⁴The country exit code is a sequence of digits that you have to dial in order to inform the telco system that you want to call a number abroad. A full list of country exit codes can be found on <http://www.howtocallabroad.com/codes.html>.

- homomorphic with respect to tenant t and function h :
 $\text{toUniversal}(h(x_1, x_2, \dots), t) = h(\text{toUniversal}(x_1, t), \text{toUniversal}(x_2, t), \dots)$

We will call a conversion function pair *fully-order-preserving* if *toUniversal* and *fromUniversal* are order-preserving with respect to all tenants. Consequently, a conversion function pair can also be *fully-h-preserving*.

Listings 6 and 7 show an exemplary conversion function pair used to convert currencies (with USD as universal format). These functions are not only equality-preserving, but also fully-SUM-preserving: as the currency conversion is nothing but a multiplication with a constant factor⁵ from `CurrencyTransform`, it does not matter in which format we sum up individual values (as long as they all have that same format). As we will see, such special properties of conversion functions are another crucial ingredient for query optimization.

```
1 CREATE FUNCTION currencyToUniversal (DECIMAL(15,2), INTEGER)
  RETURNS DECIMAL(15,2)
2 AS 'SELECT CT_to_universal*$1 FROM Tenant, CurrencyTransform
  WHERE T_tenant_key = $2 AND T_currency_key =
  CT_currency_key;'
3 LANGUAGE SQL IMMUTABLE;
```

Listing 6: Converting a currency to universal form (USD), PostgreSQL syntax

```
1 CREATE FUNCTION currencyFromUniversal (DECIMAL(15,2), INTEGER)
  RETURNS DECIMAL(15,2)
2 AS 'SELECT CT_from_universal*$1 FROM Tenant,
  CurrencyTransform WHERE T_tenant_key = $2 AND
  T_currency_key = CT_currency_key;'
3 LANGUAGE SQL IMMUTABLE;
```

Listing 7: Converting from USD to a specific currency, PostgreSQL syntax

The conversion function examples shown in Listings 4 to 7 assume the existence of tables holding additional conversion information (`CurrencyTransform` and `PhoneTransform`) as well as a table with references into these tables (named `Tenants` table). The way how a tenant can define her portion of the conversion functions is then simply to choose a specific currency and phone format as part of an initial setup procedure. However, this is only one possible implementation. MTSQL does not make any assumptions or restrictions on the implementation of conversion function pairs themselves, as long as they satisfy the properties given in Definition 2.1.

MTSQL is not the first work that talks about conversion functions. In fact, there is an entire line of work that deals with data integration and in particular with schema mapping techniques [11, 23, 25]. These works mention and take into account conversion functions, like for example a multiplication or a division by a constant. More complex conversion functions, including regular-expression-based substitutions and other arithmetic operations, can be found in *Potter’s Wheel* [41] where *conversion* is referred to as *value translation*. All these different conversion functions can potentially also be used in MTSQL which is, to the best of our knowledge, the first work that formally defines and categorizes conversion functions according to their properties.

2.2.3 Integrity Constraints. MTSQL allows for *global* integrity constraints that every tenant has to adhere to (with respect to the entirety of her data) as well as *tenant-specific* integrity constraints (that tenants can additionally impose on their own

⁵We are aware of the fact that currency conversion is not at all constant, but depends on rapidly changing exchange rates. However, we want to keep the examples as simple as possible in order to illustrate the underlying concepts. However, the general ideas of this paper also apply to temporal databases.

data). An example of a *global* referential integrity constraint is shown in the end of Listing 3. This constraint means that for every tenant, for each entry of `E_role_id`, a corresponding entry `R_role_id` has to exist in `Roles` and must be owned by that same tenant. Consider for example employee *John* with `R_role_id 0`. The constraint implies that their must be a *role 0* owned by tenant 0, which in that case is *PhD student*. If the constraint were only *tenant-specific* for tenant 1, John would not link to roles and `E_role_id 0` would just be an arbitrary numerical value. In order to differentiate *global* from *tenant-specific* constraints, the scope is used.⁶

2.2.4 Other DDL Statements. `CREATE VIEW` statements look the same as in plain SQL. As for the other DDL statements, anyone with the necessary privilege can define global views on *global* and *tenant-specific* tables. Tenants are allowed to create their own, tenant-specific views (using the default scope). The selected data has to be presented in universal format if it is a *global* view and in the *tenant-specific* format otherwise. `DROP VIEW`, `DROP TABLE` and `ALTER TABLE` work the same way as in plain SQL.

2.3 Data Control Language

Let us have a look at the MTSQL `GRANT` statement:

```
GRANT <privileges> ON <database|table> TO <ttid>;
```

Listing 8: MTSQL GRANT syntax

As in plain SQL, this grants some set of access privileges (`READ`, `INSERT`, `UPDATE` and/or `DELETE`) to the tenant identified by *ttid*. In the context of MTSQL, however, this means that the privileges are granted with respect to *C*. Consider the following statement:

```
GRANT READ ON Employees TO 42;
```

Listing 9: Example of an MTSQL GRANT statement

In the *private* table layout, if *C* is 0, then this would grant tenant 42 read access to `Employees_0`, but if *C* is 1, tenant 42 would get read access to `Employees_1` instead. If a grant statement grants to *ALL*, then the grant semantics also depend on *D*, more concretely if $D = \{7, 11, 15\}$ the privileges would be granted to tenants 7, 11 and 15.

By default, a new tenant that joins an MTSQL system is granted the following privileges: `READ` access to global tables, `READ`, `INSERT`, `UPDATE`, `DELETE`, `GRANT` and `REVOKE` on his own instances of tenant-specific tables. In our example, this means that a new tenant 111 can read and modify data in `Employees_111` and `Roles_111`. Next, a tenant can start asking around to get privileges on other tenants' tables or also on global tables. The `REVOKE` statement, as in plain SQL, simply revokes privileges that were granted with `GRANT`.

2.4 Query Language

Just as in FlexScheme [11, 12], queries themselves are written in plain SQL and have to be filtered according to *D*. Whereas in FlexScheme *D* always equals $\{C\}$ (a tenant can only query her own data), MTSQL allows cross-tenant query processing, which means that the data set can include other tenants than *C* and can in particular contain more than one element. As mentioned in the

⁶Remembering that an empty IN list refers all tenants, this is exactly what is used to indicate a global constraint. Additionally, all constraints created as part of a `CREATE TABLE` statement are global as well.

introduction, this creates some new challenges that have to be handled with special care.

2.4.1 Client Presentation. As soon as tenants can query other tenants' data, the MTSQL engine has to be make sure to deliver results in the proper format. For instance, looking again at Figure 2, if tenant 0 queries the average salary of all employees of tenant 1, then this should be presented in USD because tenant 0 stores her own data in USD and expects other data to be in USD as well. Consequently, if tenant 1 would ask that same query, the result would be returned as is, namely in EUR.

2.4.2 Comparisons. Consider a join of `Roles` and `Employees` on `role_id`. As long as the dataset size is only one, such a join query has the same semantics as in plain SQL (or FlexScheme). However, as soon as tenant 1, for instance, asks this query with $D = \{0, 1\}$, the join has to take the *ttids* into account. The reason for this is that `role_id` is a *tenant-specific* attribute and should hence only be joined within the same tenant in order to prevent semantically wrong results like John being an intern (although tenant 0 does not have such a role) or Nancy being a professor (despite the fact that tenant 1 only has roles *intern*, *researcher* and *executive*).

Comparison or join predicates containing *comparable* and *convertible* attributes, on the other hand, just have to make sure that all data is brought into universal format before being compared. For instance, if tenant 0 wants to get the list of all employees (of both tenants) that earn more than 100K USD, all employee salaries have to be converted to USD before executing the comparison.

Finally, MTSQL does not allow to compare *tenant-specific* with other attributes. For instance, we see no way how it could make sense to compare `E_role_id` to something like `E_age` or `E_salary`.

2.5 Data Manipulation Language

MTSQL DML works the same way as in FlexScheme [11, 12] if $D = \{C\}$. Otherwise, if $D \neq \{C\}$, the semantics of a DML statement are defined such that it is applied to each tenant in *D* separately. Constants, `WHERE` clauses and sub-queries are interpreted with respect to *C*, exactly the same way as for queries (c.f. Section 2.4). This implies that executing `UPDATE` or `INSERT` statements might involve value conversion to the proper tenant format(s).

3 MTBASE

Based on the concepts described in the previous section, we implemented MTBase, an open-source MTSQL engine [1]. As shown in Figure 3, the basic building block of MTBase is an MTSQL-to-SQL translation middleware sitting between a traditional DBMS and the client. In fact, as it communicates to the DBMS (and to the client) by the means of pure SQL, MTBase works in conjunction with any off-the-shelve DBMS. For performance reasons, the proxy maintains a cache of MT-specific meta data, which is persisted in the DBMS along with the actual user data. Conversion functions are implemented as UDFs that might involve additional meta tables, both of which are also persisted in the DBMS. MTBase implements the *basic data layout*, which means that *data ownership* is implemented as an additional (meta) *ttid* column in each *tenant-specific* table as illustrated in Figure 2). There are some dedicated meta tables: `Tenant` stores each tenant's privileges and conversion information and `Schema` stores information about table and attribute comparability. Additional

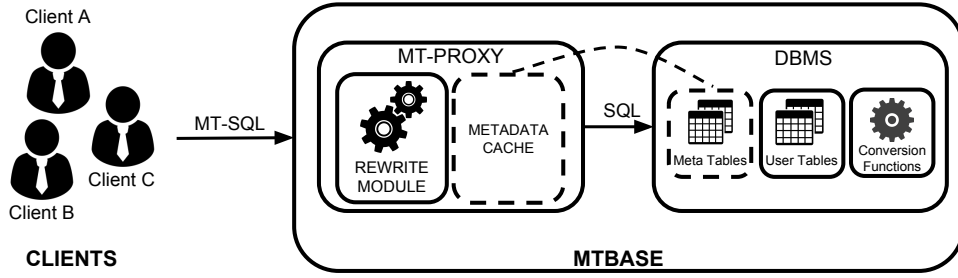


Figure 3: MTBase architecture

meta tables can (but do not have to) be used to implement conversion function pairs, as for example `CurrencyTransform` and `PhoneTransform` shown in Listings 4 to 7.

While the rewrite module was implemented in Haskell and compiled with GHC [6], the connection handling and the meta data cache maintenance was written in Python (and run with the Python2 interpreter) [4]. Haskell is handy because we can make full use of pattern matching and additive data types to implement the rewrite algorithm in a quick and easy-to-verify way, but any other functional language, like e.g. Scala [5], would also do the job. Likewise, there is nothing fundamental in using Python, any other framework that has a good-enough abstraction of SQL connections, e.g. JDBC [7], could be used.

Upon opening a connection at the middleware, the client’s *ttid*, *C*, is derived from the connection string and used throughout the entire lifetime of that connection. Whenever a client sends a MTSQL statement *s*, first if the current scope is complex, a SQL query *q_s* is derived from this scope and evaluated at the DBMS in order to determine the relevant dataset *D*. After that, *D* is compared against privileges of *C* in the `Tenant` table and *ttids* in *D* without the corresponding privilege are pruned, resulting in *D'*. Next, *C*, *D'* and *s* are input into the rewrite algorithm which produces a rewritten SQL statement *s'* which is then sent to the DBMS before relaying the result back to the client. Note that in order to guarantee correctness in the presence of updates, *q_s* and *s'* have to be executed within the same transaction and with a consistency level at least *repeatable-read* [13] (even if the client does not impose any transactional guarantees). If *s* is a DDL statement, the middleware also updates the MT-Specific meta information in the DBMS and the cache.

The rest of this section explains the MTSQL-to-SQL rewrite algorithm in its canonical form and proves its correctness with respect to Section 2.4, while Section 4 shows how to optimize the rewritten queries such that they can be run on the DBMS with reasonable performance.

3.1 Canonical Query Rewrite Algorithm

Our proposed canonical MTSQL-to-SQL rewrite algorithm works top-down, starting with the outer-most SQL query and recursively rewriting sub-queries as they come along. For each sub-query, the SQL clauses are rewritten one-by-one. The algorithm makes sure that for each sub-query the following invariant holds: the result of the sub-query is filtered according to *D'* and presented in the format required by *C*. Note that this invariant also helps to formally prove the correctness of the rewrite algorithm as we will show in Section 3.2.

The pseudo code of the general rewrite algorithm for rewriting a (sub-) query is shown in Algorithm 1. Note that `FROM`, `GROUP BY`, `ORDER BY` and `HAVING` clause can be rewritten without any

additional context while `SELECT` and `WHERE` need the whole query as an input because they might need to check the `FROM` for additional information, for instance they must know to which original tables certain attributes belong.

```

1: Input: C: ttid, D: set of ttids, Q: MTSQL query
2: Output: SQL query
3: function REWRITEQUERY(C, D, Q)
4:   new-select ← rewriteSelect(C, D, Q)
5:   new-from ← rewriteFrom(C, D, Q.from())
6:   new-where ← rewriteWhere(C, D, Q)
7:   new-group-by ← rewriteGroupBy(C, D, Q.groupBy())
8:   new-order-by ← rewriteOrderBy(C, D, Q.orderBy())
9:   new-having ← rewriteHaving(C, D, Q.having())
10:  return new Query (new-select, new-from, new-where,
    new-group-by, new-order-by, new-having)

```

Algorithm 1: Canonical Query Rewrite Algorithm

In the following, we will look at the rewrite functions for the different SQL clauses. Because of space constraints, we only provide the high-level ideas and illustrate them with suitable minimal examples. However, we strongly encourage the interested reader to check-out the Haskell code [2] which in fact almost reads like a mathematical definition of the rewrite algorithm.

SELECT The rewritten `SELECT` clause has to present every attribute *a* in *C*’s format, which, if *a* is convertible, is achieved by two calls to the conversion function pair of *a* as can be seen in the examples of Listing 10 where `-->` simply denotes rewriting. If *a* is part of compound expression (as in line 6), it has to be converted before the functions (in that case `AVG`) are applied. Note that in order to make a potential super-query work correctly, we also rename the result of the conversion, either by the new name that it got anyway (as in line 6) or by the name that it had before (as in line 3). Rewriting a star expression (line 9) in the uppermost query also needs special attention, in order not to provide the client with confusing information, like *ttids* which should stay invisible.

```

1 -- Rewriting a simple select expression:
2 SELECT E_salary FROM Employees; -->
3 SELECT currencyFromUniversal(currencyToUniversal(E_salary, ttid)
  , C) as salary FROM Employees;
4 -- Rewriting an aggregated select expression
5 SELECT AVG(E_salary) as avg_sal FROM Employees; -->
6 SELECT AVG(currencyFromUniversal(currencyToUniversal(E_salary,
  ttid), C)) as avg_sal FROM Employees;
7 -- Rewriting star expression, hiding irrelevant info
8 SELECT * FROM Employees; -->
9 SELECT E_name, E_salary, E_age FROM Employees;

```

Listing 10: Examples for Rewriting `SELECT` clause

WHERE There are essentially three steps that the algorithm has to perform in order to create a correctly rewritten `WHERE` clause (as shown in Listing 11). First, conversion functions have to be

added to each convertible attribute in each predicate in order make sure that comparisons are executed in the correct (client) format (lines 2 to 6). This happens the same way as for a SELECT clause. Notably, all constants are always in C 's format because it is C who asks the query. Second, for every predicate involving two or more tenant-specific attributes, additional predicates on $ttid$ have to be added (line 9), unless if the attributes are part of the same table, which means they are owned by the same tenant anyway. Predicates that contain `tenant-specific` together with other attributes cause the entire query to be rejected as was required in Section 2.4.2. Last, but not least, for every base table in the FROM clause, a so-called D-filter has to be added to the WHERE clause (line 12). This filter makes sure that only the relevant data (data that is owned by a tenant in D') gets processed.

```

1 -- Comparison with a constant:
2 .. FROM Employees WHERE E_salary > 50K -->
3 .. WHERE currencyFromUniversal(currencyToUniversal(E_salary,ttid
   ),C) > 50K) ..
4 -- General comparison:
5 .. FROM Employees E1, Employees E2 WHERE E1.E_salary > E2.
   E_salary -->
6 .. WHERE currencyFromUniversal(currencyToUniversal(E1.E_salary,
   E1.ttid),C) > currencyFromUniversal(currencyToUniversal(E1.
   E_salary,E1.ttid),C) ..
7 -- Extend with predicate on ttid
8 .. FROM Employees, Roles WHERE E_role_id = R_role_id -->
9 .. FROM Employees, Roles WHERE E_role_id = R_role_id AND
   Employees.ttid = Roles.ttid ..
10 -- Adding D-filters for D' = {3,7}
11 .. FROM Employees E, Roles R .. -->
12 .. WHERE E.ttid IN (3,7) AND R.ttid IN (3,7) ..

```

Listing 11: Examples for Rewriting WHERE clause

FROM All tables referred by the FROM clause are either base tables or temporary tables derived from a sub-query. Rewriting the FROM clause simply means to call the rewrite algorithm on each referenced sub-query as shown in Algorithm 2. A FROM table might also contain a JOIN of two tables (sub-queries). In that case, the two sub-queries are rewritten and then the join predicate is rewritten in the exact same way like any WHERE.

Notably, this algorithm preserves the desired invariant for (sub-) queries: the result of each sub-query is in client format and filtered according to D' , and, due to the rewrite of the SELECT and the WHERE clause of the current query, base tables, as well as joins, are also presented in client format and filtered by D . We conclude that the result of the current query therefore also preserves the invariant.

```

1: Input:  $C$ :  $ttid$ ,  $D$ : set of  $ttids$ ,
2:  $FromClause$ : MTSQL FROM clause
3: Output: SQL FROM clause
4: function REWRITEFROM( $C, D, FromClause$ )
5:    $res \leftarrow \text{extractBaseTables}(FromClause)$ 
6:   for all  $q \in \text{extractSubQueries}(FromClause)$  do
7:      $res \leftarrow res \cup \{ \text{rewriteQuery}(C, D, q) \}$ 
8:   for all  $(q_1, q_2, cond) \in \text{extractJoins}(FromClause)$  do
9:      $q'_1 \leftarrow \text{rewriteQuery}(C, D, q_1)$ 
10:     $q'_2 \leftarrow \text{rewriteQuery}(C, D, q_2)$ 
11:     $cond' \leftarrow \text{rewriteWhere}(C, D, cond)$ 
12:     $res \leftarrow res \cup \{ \text{createJoin}(q'_1, q'_2, cond') \}$ 
return  $res$ 

```

Algorithm 2: Rewrite Algorithm for FROM clause

GROUP-BY, ORDER-BY and HAVING HAVING and GROUP-BY clauses are basically rewritten the same way like the expressions in the SELECT clause. Some DBMSs might throw a

warning stating that grouping by a comparable attribute a is ambiguous because the way we rewrite a in the WHERE clause and rename it back to a , we could actually group by the original or by the converted attribute a . However, the SQL standard clearly says that in such a case, the result should be grouped by the outer-more expression, which is exactly what we need. ORDER-BY clauses need not be rewritten at all.

SET SCOPE Simple scopes do not have to be rewritten at all. The FROM and WHERE clause of a complex scope are rewritten the same way as in a sub-query. In order to make it a valid SQL query, the rewrite algorithm adds a SELECT clause that projects on the respective $ttids$ as shown in Listing 12.

```

1 SET SCOPE = "FROM Employees WHERE E_salary > 180K"; -->
2 SELECT ttid FROM Employees WHERE currencyFromUniversal(
   currencyToUniversal(E_salary,ttid),C) > 180K;

```

Listing 12: Rewriting a complex SCOPE expression

3.2 Algorithm Correctness

PROOF. We prove the correctness of the canonical rewrite algorithm with respect to Section 2.4 by induction over the composable structure of SQL queries and by showing that the desired invariant (the result of each sub-query is filtered according to D' and presented in the format required by C) holds: First, as a base, we state that adding the D-filters in the WHERE clause and transforming the SELECT clause to client format for every base table in each lowest-level sub-query ensures that the invariant holds. Next, as an induction step, we state that the way how we rewrite the FROM clause, as it was described earlier, preserves that property. The top-most SQL query is nothing but a composition of sub-queries (and base tables) for which the invariant holds. This means that the invariant holds for the entire query, which is hence guaranteed to deliver the correct result. \square

3.3 Rewriting DDL and DML Statements

Rewriting DDL and DML statements is very similar to rewriting queries, in fact, predicates are rewritten in exactly the same way. The remaining questions are how to rewrite *tenant-specific* referential integrity constraints (using check constraints) and how to apply DML statements to a dataset $D \neq \{C\}$ (by executing the proper value transformations separately for each client). While the semantics and the intuition how to implement them should be clear, we refer again to the extended version of this paper [15] for further examples and explanations.

4 OPTIMIZATIONS

As we have seen, there is a canonical rewrite algorithm that correctly rewrites MTSQL to SQL. However, we will show in Section 5 that the rewritten queries often execute very slowly on the underlying DBMS. The main reason for this is that the pure rewritten queries call two conversion functions on every transformable attribute of every record that is processed, which is extremely expensive. Luckily, the execution costs can be reduced dramatically when applying the optimization passes that we describe in this section. As we assume the underlying DBMS to optimize query execution anyway, we focus on optimizations that a DBMS query optimizer cannot do (because it needs MT-specific context) or does not do (because an optimization is not frequent enough outside the context of MTBase). We differentiate between *semantic optimizations*, which are always applied because they never

make a query slower and *cost-based* optimizations which are only applied if the predicted costs are smaller than in the original query.

```

1 -- dropping D-filter if D is the empty scope:
2 SELECT E_age FROM Employees WHERE E_ttid IN (1,2); -->
3 SELECT E_age FROM Employees;
4 -- dropping ttid from join predicate if |D| = 1:
5 SELECT E_age, R_name FROM Employees, Roles WHERE E_role_id =
   R_role_id AND E_ttid = R_ttid AND E_ttid IN (2) AND R_ttid
   IN (2); -->
6 SELECT E_age, R_name FROM Employees, Roles WHERE E_role_id =
   R_role_id AND E_ttid IN (2) AND R_ttid IN (2);
7 -- dropping conversion functions if D = {C}:
8 SELECT currencyFromUniversal(currencyToUniversal(E_salary,
   E_ttid),0) AS E_salary FROM Employees; -->
9 SELECT E_salary FROM Employees;

```

Listing 13: Examples for trivial semantic optimizations

4.1 Trivial Semantic Optimizations

There are a couple of special cases for C and D that allow to save conversion function calls, join predicates and/or D -filters. First, if D includes all tenants, that means that we want to query all data and hence D -filters are no longer required as shown in line 3 of Listing 13. Second, as shown in line 6, if $|D| = 1$, we know that all data is from the same tenant, which means that including $ttid$ in the join predicate is no longer necessary. Last, if we know that a client queries her own data, i.e. $D = \{C\}$ corresponds to the default scope, we know that even convertible attributes are already in the correct format and can hence remove the conversion function calls (line 9).

4.2 Other Semantic Optimizations

There are a couple of other semantic optimizations that can be applied to rewritten queries. While *client presentation push-up* and conversion push-up minimize the number of conversions by delaying conversion to the latest possible moment, *aggregation distribution* takes into account specific properties of conversion functions (as mentioned in Section 2.2.2). If conversion functions are UDFs written in SQL it is also possible to inline them. This typically gives queries an additional speed up.

```

1 -- before optimization
2 SELECT Dom.name1, Dom.sal1 as sal, COUNT(*) as cnt FROM (
3   SELECT E1.name as name1, currencyFromUniversal(
4     currencyToUniversal(E1.E_salary, E1.E_ttid), C) as sal1
5   FROM Employees E1, Employees E2
6   WHERE currencyFromUniversal(currencyToUniversal(E1.E_salary,
7     E1.E_ttid), C) >
8     currencyFromUniversal(currencyToUniversal(E2.E_salary, E2.
9     E_ttid), C)
10 ) as Dom GROUP BY Dom.name1, sal, cnt ORDER BY cnt;
11 -- after optimization
12 SELECT Dom.name1, currencyFromUniversal(Dom.sal1, C) as sal,
13   COUNT(*) as cnt FROM (
14   SELECT E1.name as name1, currencyToUniversal(E1.E_salary, E1.
15     E_ttid) as sal1
16   FROM Employees E1, Employees E2
17   WHERE currencyToUniversal(E1.E_salary, E1.E_ttid) >
18     currencyToUniversal(E2.E_salary, E2.E_ttid)
19 ) as Dom GROUP BY Dom.name1, sal, cnt ORDER BY cnt;

```

Listing 14: Example for client presentation push-up

4.2.1 Client Presentation and Conversion Push-Up. As conversion function pairs are equality-preserving, it is possible in some cases to defer conversions to later, for example to the outermost query in the case of nested queries. While *client presentation push-up* converts everything to universal format and defers conversion to client format to the outermost SELECT clause, *conversion push-up* pushes this idea even more by also delaying the conversion to universal format as much as possible. Both optimizations are beneficial if the delaying of conversions allows the query execution engine to evaluate other (less expensive) predicates first. This means that, once the data has to be converted, it is already

more filtered and therefore the overall number of (expensive) conversion function calls becomes smaller (or, in the worst case, stays the same). Naturally, if we delay conversion, this also means that we have to propagate the necessary *ttids* to the outer-more queries and keep track of the current data format.

Listing 14 shows a query that ranks employees according to the fact how many salaries of other employees their own salary dominates. With *client presentation push-up*, salaries are compared in universal instead of client format, which is correct because of the equality-preserving property (c.f. Corollary 1) and saves half of the function calls in the sub-query.

Conversion push-up, as shown in Listing 15, reduces the number of function calls dramatically: First, as it only converts salaries in the end, salaries of employees aged less than 45 do not have to be considered at all. Second, the WHERE clause converts the constant (100K) instead of the attribute (E_salary). As the outcome of conversion functions is immutable (c.f. Section 2.2.2) and C is also constant, the conversion functions have to be called only once per tenant and are then cached by the DBMS for the rest of the query execution, which becomes much faster as we will see in Section 5.

```

1 -- before optimization
2 SELECT AVG(X.sal) FROM (
3   SELECT currencyFromUniversal(currencyToUniversal(E_salary,
4     E_ttid), C) as sal
5   FROM Employees WHERE E_age >= 45 AND
6     currencyFromUniversal(currencyToUniversal(E_salary, E_ttid), C)
7     > 100K) as X;
8 -- after optimization
9 SELECT AVG(currencyFromUniversal(currencyToUniversal(X.sal, X.
10   sal_ttid),C)) FROM (
11   SELECT E_salary as sal, E_ttid as sal_ttid
12   FROM Employees WHERE E_age >= 45 AND
13     E_salary > currencyFromUniversal(currencyToUniversal(100K,
14     E_ttid), C) as X);

```

Listing 15: Example for conversion push-up

4.2.2 Aggregation Distribution. Many analytical queries contain aggregation functions, some of which aggregate on *convertible* attributes. The idea of aggregation distribution is to aggregate in two steps: First, aggregate per tenant in that specific tenant format (requires no conversion) and second, convert intermediary results to universal (one conversion per tenant), aggregate those and convert the final result to client format (one additional conversion). This simple idea reduces the number of conversion function calls for N records and T different data owners of these records from $(2N)$ to $(T + 1)$. This is significant because T is typically much smaller than N (and cannot be greater).

Compared to pure *conversion push-up*, which works for any conversion function pair, the applicability of *aggregation distribution* depends on further algebraic properties of these functions. Gray et al. [24] categorize numerical aggregation functions into three categories with regard to their ability to distribute: *distributive* functions, like COUNT, SUM, MIN and MAX distribute with functions F (for partial) and G (for total aggregation). For COUNT for instance, F is COUNT and G is SUM as the total count is the sum of all partial counts. There are also *algebraic* aggregation functions, e.g. AVG, where the partial results are not scalar values, but tuples. In the case of AVG, this would be the pairs of a partial sums and partial counts because the total average can be computed from the sum of all sums, divided by the sum of all counts. Finally, *holistic* aggregation functions cannot be distributed at all.

We would like to extend the notion of Gray et al. [24] and define the *distributability of an aggregation function a with respect to a conversion function pair $(from, to)$* . Table 2 shows some examples for different aggregation and conversion functions. First

	$to(x) = c \cdot x$	$to(x) = a \cdot x + b$	$to = \text{order-preserving}$	$to = \text{equality-preserving}$
COUNT	✓	✓	✓	✓
MIN	✓	✓	✓	✗
MAX	✓	✓	✓	✗
SUM	✓	✓	✗	✗
AVG	✓	✓	✗	✗
<i>Holistic</i>	✗	✗	✗	✗

Table 2: Distributability of different aggregation functions over different categories of conversion functions

of all, we want to state that, as all conversion functions have scalar values as input and output, they are always fully-COUNT-preserving, which means that COUNT can be distributed over all sorts of conversion functions. Next, we observe that all *order-preserving functions* preserve the minimum and the maximum of a given set of numbers, which is why MIN and MAX distribute over the first three categories of conversion functions displayed in Table 2. We further notice that if *to* (and consequently also *from*) is a multiplication with a constant (first column of Table 2), *to* is fully-MIN-, fully-MAX- and fully-SUM-preserving, which is why these aggregation functions distribute. As SUM and COUNT distribute, AVG, an algebraic function, distributes as well.

Finally looking at the second column of Table 2, we see that even linear functions are SUM- and AVG-preserving. To see why, we can think about computing the average over all tenants as a weighted average of partial (per-tenant) averages for AVG and multiply these partial averages with the partial counts to reconstruct the total sum [15, Appendix B].

```

1 -- before optimization
2 SELECT SUM(currencyFromUniversal(currencyToUniversal(E_salary,
   E_ttid), C)) as sum_sal FROM Employees
3 -- after optimization
4 SELECT currencyFromUniversal(SUM(t.E_partial_salary), C) as
   sum_sal FROM (SELECT currencyToUniversal(SUM(E_salary),
   E_ttid) as E_partial_salary FROM Employees GROUP BY E_ttid)
   as t;

```

Listing 16: Example for conversion function distribution

We conclude this subsection by observing that the conversion function pair for *phone format* (c.f. Listings 4 and 5) is not even *order-preserving* and does therefore not distribute while the pair for *currency format* (c.f. Listings 6 and 7) distributes over all standard SQL aggregation functions. An example of how this can be used is shown in Listing 16.

```

1 -- before optimization
2 SELECT currencyFromUniversal(currencyToUniversal(E_salary,
   E_ttid), C) as E_salary FROM Employees
3 -- after optimization
4 SELECT (C1.CT_from_universal * C2.CT_to_universal * E_salary) as
   E_salary
5 FROM Employees, Tenant T1, Tenant T2, CurrencyTransform1,
   CurrencyTransform2
6 WHERE T1.T_tenant_key = C AND T1.T_currency_key =
   CurrencyTransform1.CT_currency_key AND
7 T2.T_tenant_key = E_ttid AND T2.T_currency_key =
   CurrencyTransform2.CT_currency_key

```

Listing 17: Example for function inlining

4.2.3 Function Inlining. As explained in Section 2.2.2, there are several ways how to define conversion functions. However, if they are defined as a SQL statement (potentially including lookups into meta tables), they can be directly inlined into the rewritten query in order to save calls to UDFs. Function inlining typically also enables the query optimizer of the underlying DBMS to optimize much more aggressively. In WHERE clauses, conversion functions could simply be inlined as sub-queries, which, however often results in sub-optimal performance as calling a sub-query on each conversion is not much cheaper than calling the corresponding UDF. For SELECT clauses, the SQL standard does anyway not allow to inline as a sub-query as this can result in attributes

not being contained neither in an aggregate function nor in the GROUP BY clause, which is why most commercial DBMS reject such queries (while PostgreSQL, for instance executes them anyway). This is why the proper way to inline functions is by using a join as shown in Listing 17. Our results in Section 5 suggest that function inlining, though producing complex-looking SQL queries, results in very good query execution performance.

It is important to mention that function inlining should only happen after the other semantic optimization passes because these other passes are able to *reduce the number* of required UDF calls, while function inlining can only make a UDF call *faster*. Furthermore, it is important to understand that, while some clever query optimizers do indeed inline UDF calls already, none of the query optimizers that we looked at seems to perform *client presentation* and *conversion push-up*, let alone *aggregation distribution*, despite the fact that the foundation for these transformations [24, 28] have been established already more than 20 years ago.

5 EXPERIMENTS AND RESULTS

This section presents the evaluation of MTBase using an extension from the well-known TPC-H benchmark [43], called *MT-H* [15]. We first evaluated the benefits of different optimization steps from Section 4 and found that the combination of all of these steps brings the biggest benefit. Second, we analyzed how MTBase scales with an increasing number of tenants. With all optimizations applied and for a dataset of 100 GB on a single machine, MTBase scales up to thousands of tenants with very little overhead. We also validated result correctness as explained in Section 5.1 and can report only positive results.

5.1 MT-H Benchmark

MT-H uses the same database schema as TPC-H, but considers the Customer, Order, and Lineitem tables *tenant-specific* and the remaining tables *global*. Attributes C_acctbal, O_totalprice, and L_extendedprice are considered *convertible* with respect to the conversion functions of Listings 6 and 7 and C_phone with respect to Listings 4 and 5. While C_custkey, O_orderkey, O_custkey, L_orderkey are *tenant-specific*, all remaining attributes are *comparable*. A detailed description on this benchmark, including the validation of query results, can be found in our technical report [15].

5.2 Setup

In our experiments, we used the following two setups: The first setup is a PostgreSQL 9.6 Beta installation, running on Debian Linux 4.1.12 on a 4x16 Core AMD Opteron 6174 processor with 256 GB of main memory. The second installation runs a commercial database (which we will call *System C*) on a commercial operating system and on the same processor with 512 GB of main memory. Although both machines have enough secondary storage capacity available, we decided to configure both database management systems to use in-memory backed files in order to achieve the best performance possible. Moreover, we configured the systems to use all available threads, which enabled *intra-query parallelism*.

5.3 Workload and Methodology

As the MT-H benchmark has a lot of parameters and in order to make things more concrete, we worked with the following two scenarios: *Scenario 1* handles the data of a business alliance of a couple of small to mid-sized enterprises, which means there are 10 tenants with $sf = 1$ and each of them owns more or less

Level	Q01	Q02	Q03	Q04	Q05	Q06	Q07	Q08	Q09	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
tpch-0.1G	2.6	0.11	0.27	0.35	0.15	0.29	0.18	0.14	0.59	0.36	0.081	0.37	0.26	0.27	0.77	0.12	0.081	0.89	0.12	0.13	0.57	0.081
canonical	84	1.0	0.55	0.65	0.32	1.0	0.29	0.36	4.9	0.91	0.37	0.55	0.63	0.98	3.1	1.2	0.49	1.7	0.3	2.8	0.66	2.0
o1	2.7	1.0	0.43	0.61	0.22	0.43	0.23	0.56	3.8	0.76	0.37	0.55	0.92	0.56	0.91	1.2	0.48	1.6	0.3	2.8	0.66	0.085
o2	2.7	1.0	0.42	0.61	0.22	0.43	0.23	0.57	3.9	0.76	0.38	0.55	0.89	0.56	0.96	1.2	0.5	1.7	0.3	2.8	0.67	0.085
o3	2.7	1.0	0.43	0.61	0.22	0.43	0.23	0.56	3.9	0.76	0.37	0.55	0.92	0.56	0.91	1.2	0.48	1.6	0.3	2.8	0.66	0.085
o4	2.7	1.0	0.43	0.62	0.22	0.43	0.23	0.61	4.1	0.78	0.39	0.56	0.9	0.57	1.0	1.2	0.51	1.7	0.31	3.1	0.67	0.085
inl-only	2.7	1.0	0.42	0.65	0.22	0.43	0.22	0.57	3.8	0.76	0.37	0.55	0.92	0.56	0.92	1.2	0.48	1.6	0.3	2.8	0.66	0.085

Table 3: Response times [sec] of 22 TPC-H queries for MTBase-on-PostgreSQL with, $sf = 1$, $T = 10$, $\rho = \text{uniform}$, $C = 1$, $D = \{1\}$, for different levels of optimizations, versus TPC-H with $sf = 0.1$

Level	Q01	Q02	Q03	Q04	Q05	Q06	Q07	Q08	Q09	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
tpch-0.1G	2.6	0.11	0.27	0.35	0.15	0.29	0.18	0.14	0.59	0.36	0.081	0.37	0.26	0.27	0.77	0.12	0.081	0.89	0.12	0.13	0.57	0.081
canonical	87	1.0	0.5	0.6	0.28	1.0	0.26	0.37	4.9	0.89	0.37	0.56	0.65	1.0	3.2	1.2	0.49	1.6	0.31	2.8	0.66	2.0
o1	87	1.0	0.5	0.69	0.33	1.0	0.27	0.38	5.2	0.9	0.39	0.56	0.92	1.0	3.1	1.2	0.51	1.6	0.32	3.1	0.68	2.0
o2	87	1.0	0.5	0.61	0.28	1.0	0.27	0.38	5.2	0.9	0.39	0.57	0.91	1.0	3.1	1.2	0.51	1.6	0.32	3.1	0.67	1.3
o3	32	1.0	0.45	0.63	0.28	0.44	0.24	0.37	4.3	0.83	0.38	0.56	0.91	1.1	1.9	1.3	0.51	1.6	0.32	3.1	0.67	1.3
o4	14	1.0	0.48	0.62	0.22	0.44	0.23	0.57	3.9	0.93	0.38	0.56	0.89	0.73	1.3	1.2	0.49	1.6	0.3	2.8	0.66	0.27
inl-only	45	1.0	0.47	0.61	0.27	0.64	0.24	0.58	4.2	0.94	0.37	0.55	0.91	0.73	2.2	1.2	0.48	1.7	0.3	2.8	0.66	0.27

Table 4: Response times [sec] of 22 TPC-H queries for MTBase-on-PostgreSQL with, $sf = 1$, $T = 10$, $\rho = \text{uniform}$, $C = 1$, $D = \{2\}$, for different levels of optimizations, versus TPC-H with $sf = 0.1$

Level	Q01	Q02	Q03	Q04	Q05	Q06	Q07	Q08	Q09	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
tpch-1G	26	1.2	4.5	1.4	1.5	2.9	3.7	1.3	9.5	2.2	0.38	3.9	8.4	2.7	5.9	1.2	0.54	10	0.3	2.4	4.8	0.47
canonical	870	1.1	6.5	1.5	3.4	8.7	3.7	1.7	19	11	0.36	4.1	4.9	7.3	28	1.2	0.57	12	0.32	2.6	5.8	20
o1	860	1.1	6.5	1.5	3.4	8.7	3.7	1.7	19	11	0.36	4.1	4.9	7.3	28	1.2	0.62	12	0.33	2.7	5.9	20
o2	870	1.1	6.5	1.5	3.4	8.6	3.7	1.7	19	11	0.35	4.1	4.9	7.2	28	1.2	0.57	12	0.32	2.6	5.8	13
o3	310	1.1	5.5	1.5	3.1	3.1	3.4	1.6	11	10	0.36	4.1	4.9	7.3	12	1.2	0.55	12	0.32	2.6	5.9	13
o4	130	1.1	3.7	1.5	1.7	3.1	3.4	1.4	11	4.6	0.38	4.1	4.9	4.4	9.1	1.2	0.59	12	0.32	2.6	5.7	2.2
inl-only	450	1.1	4	1.6	1.8	5.1	3.5	1.4	14	4.9	0.39	4.1	4.8	4.4	19	1.2	0.55	12	0.32	2.6	5.8	2.3

Table 5: Response times [sec] of 22 TPC-H queries for MTBase-on-PostgreSQL with $sf = 1$, $T = 10$, $\rho = \text{uniform}$, $C = 1$, $D = \{1, 2, \dots, 10\}$, for different levels of optimizations, versus TPC-H with $sf = 1$

the same amount of data ($\rho = \text{uniform}$). *Scenario 2* simulates the HDC use case [27] and hence needs to be is a huge database ($sf = 100$) of medical records coming from thousands of tenants, like hospitals and private practices. Some of these institutions have vast amounts of data while others only handle a couple of patients ($\rho = \text{zipf}$). A research institution wants to query the entire database ($D = \{1, 2, \dots, T\}$) in order to gather new insights for the development of a new treatment. We looked at this scenario for different numbers of T .

In order to evaluate the overhead of *cross-tenant query processing* in MTBase compared to single-tenant query processing, we also measured the standard TPC-H queries with different scaling factors. When D was set to all tenants, we compared to TPC-H with the same scaling factor as MT-H. For the cases where D had only one tenant (out of ten), we compared with TPC-H with a scaling factor ten times smaller.

Every query run was repeated three times in order to ensure stable results. We noticed that three runs are needed for the response times to converge (within 2%). Thus we always report the last measured response time for each query with two significant digits.

All experiments were executed with both setups (PostgreSQL and *System C*). Whereas the major findings were the same on both systems, PostgreSQL optimizes conversion functions (UDFs) much better by caching their results. *System C*, on the other hand does not allow UDFs to be defined as deterministic and hence cannot cache conversion results. This eliminates the effect of *conversion push-up* when applied to comparison predicates where we convert the constant instead of the attribute (c.f. Listing 15). This being said, the rest of this section only reports results on PostgreSQL while we encourage the interested reader to also consult our additional results [15] to confirm that the main conclusions drawn from the PostgreSQL experiments generalize.

opt level	optimization passes
canonical	none
o1	trivial optimizations
o2	o1 + client presentation push-up + conversion push-up
o3	o2 + conversion function distribution
o4	o3 + conversion function inlining
inl-only	o1 + conversion function inlining

Table 6: Different optimization levels for evaluation

5.4 Benefit of Optimizations

In order to test the benefit of the different combinations of optimizations applied, we tested *Scenario 1* with different optimization levels as shown in Table 6. From *o1* to *o4* we added optimizations incrementally, while the last optimization level (*inl-only*) only applied trivial optimizations and function inlining in order to test whether the other optimizations are useful at all.

Table 3 shows the MT-H queries for different optimization levels and *Scenario 1* ($sf = 1$, $T = 10$) where client 1 queries her own data. As we can see, in that case, applying trivial optimizations in *o1* is enough because these already eliminate all conversion functions and joins and only the D-filters remain. Executing these filters seems to be very inexpensive because most response times of the optimized queries are close to the baseline, TPC-H with $sf = 0.1$. Queries 2, 11 and 16 however, take roughly ten times longer than the baseline. This is not surprising when taken into account that these queries only operate on *shared tables* which have ten times more data than in TPC-H. The same effect can be observed in Q09 where a significant part of the joined tables are shared.

Table 4 shows similar results, but for $D = 2$, which means that now conversion functions can no longer be optimized away.

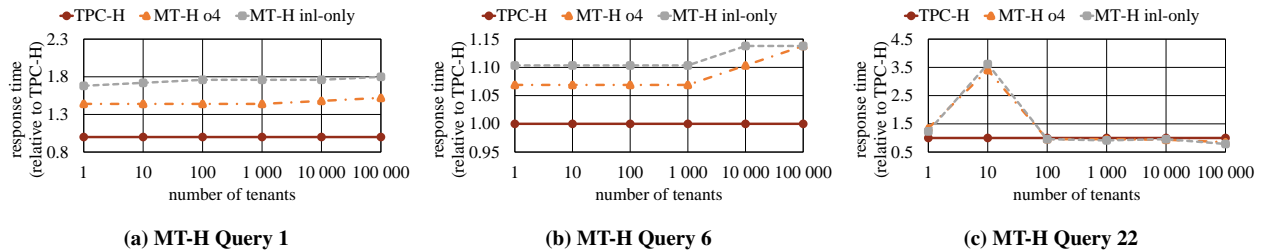


Figure 4: Response times (relative to TPC-H) of *o4* and *inlining-only* optimization levels for selected MT-H queries, $sf = 100$, T scaling from 1 to 100,000 on a log-scale, MTBase-on-PostgreSQL

While most of the queries show a similar behaviour than in the previous experiment, for the ones that involve a lot of conversion functions (i.e. queries 1, 6 and 22), we see how the performance becomes better with each optimization pass added. We also notice that while function inlining is very beneficial in general, it is even more so when combined with the other optimizations.

Finally, Table 5 shows the results where we query all data, i.e. $D = \{1, 2, \dots, 10\}$. This experiment involves even more conversion functions from all the different tenant formats into universal. In particular, when looking again at queries 1, 6 and 22, we observe the great benefit of *conversion function distribution* (added with optimization level *o3*), which, in turn, only works as great in conjunction with *client and conversion function push-up* because aggregation typically happens in the outermost query while conversion happens in the sub-queries. Overall, *o4*, which contains all optimization passes that MTBase offers, is the clear winner.

5.5 Cross-Tenant Query Processing at Large

In our final experiment, we evaluated the cost of *cross-tenant query processing* up to thousands of tenants. More concretely, we measured the response time of conversion-intensive MT-H queries (queries 1, 6 and 22) for a varying number of tenants between 1 and 100,000, for a large dataset where $sf = 100$ and for the best optimization level (*o4*) as well as for *inlining-only*. The obtained results were then compared to plain TPC-H with $sf = 100$, as shown in Figure 4. First of all, we notice that the cost overhead compared to *single-tenant query-processing* (TPC-H) stays below a factor of 2 and in general increases very moderately with the number of tenants. An interesting artifact can be observed for query 22 where MT-H for one tenant executes faster than plain TPC-H. The reason for this is a sub-optimal optimization decision in PostgreSQL: one of the most expensive parts of query 22, namely to find customers with a specific country code, is executed with a parallel scan in MT-H while no parallelism is used in the case of TPC-H.

6 RELATED WORK

MTBase builds heavily on and extends a lot of related work. This section gives a brief summary of the most prominent lines of work that influenced our design.

Data Integration Data integration (DI) is generally about finding schema and data mappings between the original schemas of different data sources and a target schema specified by the client application [23, 25, 41]. As such, DI techniques are applicable to the entire spectrum of multi-tenant databases because even if tenants use different schemas or databases, these techniques can identify correlations and hence extract useful information. Our work embraces and builds on top of the latest DI work, solving

the DI problem very efficiently for a specific case (*SS* and *ST*). More concretely, we automatically determine join predicates from schema meta data and optimize conversion functions similar to those used in DI by thoroughly analyzing and exploiting their algebraic properties. In addition, instead of translating data into a specific client format (and update periodically), we convert it to any required client format efficiently and *just-in-time*.

Database Federation: DI is often combined with database federation [26, 31], which means that there exist small program modules (called *integrators*, *mediators* or simply *wrappers*) to map data from different sources (possibly not all of them SQL databases) into one common format. While data federation generalizes well across the entire spectrum of multi-tenant databases, maintaining such wrapper architectures is expensive, both in terms of code maintenance and update processing. Conversely, MTSQL enables cross-tenant query processing in a more efficient and flexible way in the context of *SS* and *ST* databases.

Data Warehousing: Another approach how data integration can happen is during extract-transform-load (ETL) operations from different (OLTP) databases into a data warehouse [29]. Data warehouses have the well-known drawbacks that there are costly to maintain and that the data is possibly outdated [10, 14, 37]. Meanwhile, MTBase was specifically designed to work well in the context of integrated OLTP/OLAP systems, also known as *hybrid transaction-analytical processing (HTAP)* systems, and could therefore be advocated as *in-situ* or *just-in-time* data integration. Another interesting approach to *just-in-time*, respectively *on-demand* data integration, are *lenses* [45] which allow to speed up ETL processes by lowering the result accuracy to the specific level required by the application.

Shared-resources (SR) systems: In related work, this approach is also often called *database virtualization* or *database as a service (DaaS)* when it is used in the cloud context. Important lines of work in this domain include (but are not limited to) *SqlVM/Azure SQL DB* [20, 36], *RelationalCloud* [35], *SAP-HANA* [42], *Snowflake* [18] and *Oracle’s multitenant container database (CDB)* [40], most of which is well summarized in [22]. *MTBase* complements these systems by providing a platform that can accommodate more, but typically smaller tenants.

Shared-databases (SD) systems: This approach, while appearing in the *spectrum of multi-tenant databases* by Chong et al. [17], is rare in practice. *Sql Azure DB* [20] seems to be the only product that has an implementation of this approach. However, even Microsoft strongly advises against using SD and instead recommends to either use SR or ST [34].

Shared-tables (ST) systems: Work in that area includes Salesforce [44], Apache Phoenix [9], FlexScheme [11, 12] and Azure SQL Database [34]. Their common idea, as in *MTSQL*, is to use an invisible *tenant identifier* to identify which records belong to which tenant and rewrite SQL queries in order to include filters on this `ttid`. *MTSQL* extends these systems by providing the necessary features for cross-tenant query processing.

Privacy/Confidentiality: Clearly, *cross-tenant query processing* almost immediately raises the question of data confidentiality. In the case of the HDC, for instance, patients might consent to their data being used in aggregated analytics, but they most certainly would not want sensitive, personal information, like their social security number, to appear in any report. While it is out of the scope of this paper to thoroughly discuss data confidentiality in a multi-tenant system, this work establishes proper syntax and semantics for *cross-tenant query processing*, which lays the ground for building appropriate encryption mechanisms [16, 21] atop as is sketched in our technical report [15].

UDFs and Complex Expressions: *Oracle MLE* [39] is a system that allows for highly-optimized execution of user-defined functions, which makes it a promising candidate to further investigate optimization of *conversion functions*. For instance, we would like to look at optimizing complex expressions, containing several nested user-defined function calls, as a whole.

7 CONCLUSION

This paper presented *MTSQL*, a new language to address *cross-tenant query processing* in multi-tenant databases. *MTSQL* extends SQL with multi-tenancy-aware syntax and semantics, which allows to efficiently optimize and execute cross-tenant queries in *MTBase*. *MTBase* is an open-source system that implements *MTSQL*. At its core, it is an *MTSQL*-to-SQL rewrite middleware sitting between a client and any multi-tenant DBMS of choice. The performance evaluation with a benchmark adapted from TPC-H showed that *MTBase* (on top of PostgreSQL) can scale to thousands of tenants at very low overhead and that our proposed optimizations to *cross-tenant queries* are highly effective.

In the future, we plan to further analyze the interplay between the *MTBase* query optimizer and its counter-part in the DBMS execution engine in order to assess the potential of cost-based optimizations. We also want to study conversion functions that vary over time and investigate how *MTSQL* can be extended to temporal databases. Moreover, we would like to look more into the privacy issues of multi-tenant databases, in particular how to enable *cross-tenant query processing* if data is encrypted.

REFERENCES

- [1] 2017. *MTBase* project page. <https://github.com/mtbase/overview>. (2017).
- [2] 2017. *MTBase* Rewrite Algorithm. <https://github.com/mtbase/mt-rewrite>. (2017).
- [3] 2017. Oracle Virtual Private Database. <http://www.oracle.com/technetwork/databases/security/index-088277.html>. (2017).
- [4] 2017. Python 2.7.2 Release. <https://www.python.org/download/releases/2.7.2>. (2017).
- [5] 2017. Scala Language. <http://www.scala-lang.org>. (2017).
- [6] 2017. The Glasgow Haskell Compiler. <https://www.haskell.org/ghc>. (2017).
- [7] 2017. The Java Database Connectivity (JDBC). <http://www.oracle.com/technetwork/java/javase/jdbc/index.html>. (2017).
- [8] Amazon Webservices. 2017. Amazon Relational Database Service (RDS). <https://aws.amazon.com/rds>. (2017).
- [9] Apache Foundation. 2017. Apache Phoenix: High performance relational database layer over HBase for low latency applicationsn - Multi-Tenancy Feature. <http://phoenix.apache.org/multi-tenancy.html>. (2017).
- [10] Joy Arulraj et al. 2016. Bridging the Archipelago between Row-Stores and Column-Stores for Hybrid Workloads. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*, Vol. 19. 57–63.
- [11] Stefan Aulbach et al. 2008. Multi-tenant databases for software as a service: schema-mapping techniques. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM, 1195–1206.
- [12] Stefan Aulbach et al. 2011. Extensibility and data sharing in evolving multi-tenant databases. In *Data engineering (icde), 2011 ieee 27th international conference on*. IEEE, 99–110.
- [13] Hal Berenson et al. 1995. A Critique of ANSI SQL Isolation Levels. *SIGMOD Rec.* 24, 2 (1995), 1–10. <http://doi.acm.org/10.1145/568271.223785>
- [14] Lucas Braun et al. 2015. Analytics in Motion: High Performance Event-Processing AND Real-Time Analytics in the Same Database. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 251–264.
- [15] Lucas Braun et al. 2017. *MTBase: Optimizing Cross-Tenant Database Queries*. *arXiv preprint arXiv:1703.04290* (2017).
- [16] Jose M Alcaraz Calero et al. 2010. Toward a Multi-Tenancy Authorization System for Cloud Services. *IEEE Security & Privacy* 8, 6 (2010), 48–55.
- [17] Frederick Chong et al. 2006. Multi-tenant data architecture. *MSDN Library, Microsoft Corporation* (2006), 14–30.
- [18] Benoît Dageville et al. 2016. The Snowflake Elastic Data Warehouse. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. 215–226. <http://doi.acm.org/10.1145/2882903.2903741>
- [19] Sudipto Das et al. 2013. ElasTraS: An elastic, scalable, and self-managing transactional database for the cloud. *ACM Transactions on Database Systems (TODS)* 38, 1 (2013), 5.
- [20] Sudipto Das et al. 2016. Automated Demand-driven Resource Scaling in Relational Database-as-a-Service. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. 1923–1934. <http://doi.acm.org/10.1145/2882903.2903733>
- [21] Sabrina De Capitani Di Vimercati et al. 2007. Over-encryption: management of access control evolution on outsourced data. In *Proceedings of the 33rd international conference on Very large data bases*. VLDB endowment, 123–134.
- [22] Aaron J Elmore et al. 2013. Towards database virtualization for database as a service. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1194–1195.
- [23] Ronald Fagin et al. 2009. Clio: Schema mapping creation and data exchange. In *Conceptual Modeling: Foundations and Applications*. Springer, 198–236.
- [24] Jim Gray et al. 1997. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data mining and knowledge discovery* 1, 1 (1997), 29–53.
- [25] L M Haas et al. 1999. Transforming heterogeneous data with database middleware: Beyond integration. *Data Engineering* (1999), 31.
- [26] Laura M Haas et al. 2002. Data integration through database federation. *IBM Systems Journal* 41, 4 (2002), 578–596.
- [27] E Hafen et al. 2014. Health data cooperativescitizen empowerment. *Methods Inf Med* 53, 2 (2014), 82–86.
- [28] Joseph M Hellerstein et al. 1993. *Predicate migration: Optimizing queries with expensive predicates*. Vol. 22. ACM.
- [29] Ralph Kimball et al. 2002. The data warehouse toolkit: the complete guide to dimensional modelling. *Nachdr.*. New York [ua]: Wiley (2002), 1–447.
- [30] SPT Krishnan et al. 2015. Google App Engine. In *Building Your Next Big Thing with Google Cloud Platform*. Springer, 83–122.
- [31] Alon Levy. 1998. The information manifold approach to data integration. *IEEE Intelligent Systems* 13, 5 (1998), 12–16.
- [32] Simon Manfred Loesing. 2015. *Architectures for elastic database services*. Ph.D. Dissertation. ETH Zürich, Diss. Nr. 22441.
- [33] Microsoft Corporation. 2017. Microsoft Azure Multi-Tenant Architecture. <https://msdn.microsoft.com/en-gb/library/hh534480.aspx>. (2017).
- [34] Microsoft Corporation. 2017. Microsoft Azure SQL Database. <https://azure.microsoft.com/en-us/services/sql-database>. (2017).
- [35] Barzan Mozafari et al. 2013. DBSeer: Resource and Performance Prediction for Building a Next Generation Database Cloud.. In *CIDR*.
- [36] Vivek R Narasayya et al. 2013. SQLVM: Performance Isolation in Multi-Tenant Relational Database-as-a-Service.. In *CIDR*.
- [37] Thomas Neumann et al. 2015. Fast serializable multi-version concurrency control for main-memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 677–689.
- [38] Oracle Corporation. 2017. ORACLE Cloud. <https://cloud.oracle.com/database>. (2017).
- [39] Oracle Corporation. 2017. ORACLE Multilingual Engine. <http://www.oracle.com/technetwork/database/multilingual-engine>. (2017).
- [40] Oracle Corporation. 2017. ORACLE Multitenant. <http://www.oracle.com/technetwork/database/multitenant>. (2017).
- [41] Vijayshankar Raman et al. 2001. Potter’s wheel: An interactive data cleaning system. In *VLDB*, Vol. 1. 381–390.
- [42] SAP, November 2014. 2017. SAP HANA SPS 09 - What’s New? https://hcp.sap.com/content/dam/website/saphana/en_us/Technology%20Documents/SPS09/SAP%20HANA%20SPS%2009%20-%20Multitenant%20Database%20Cont.pdf. (2017).
- [43] Transaction Processing Council. 2017. TPC-H. <http://www.tpc.org/tpch>. (2017).
- [44] Craig D Weissman et al. 2009. The design of the force. com multitenant internet application development platform.. In *SIGMOD Conference*. 889–896.
- [45] Ying Yang et al. 2015. Lenses: An on-demand approach to etl. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1578–1589.

Extending In-Memory Relational Database Engines with Native Graph Support

Mohamed S. Hassan
Purdue University
West Lafayette, IN
msaberab@cs.purdue.edu

Tatiana Kuznetsova
Purdue University
West Lafayette, IN
tkuznets@cs.purdue.edu

Hyun Chai Jeong
Purdue University
West Lafayette, IN
jeong3@cs.purdue.edu

Walid G. Aref
Purdue University
West Lafayette, IN
aref@cs.purdue.edu

Mohammad Sadoghi
University of California
Davis, CA
msadoghi@ucdavis.edu

ABSTRACT

The plethora of graphs and relational data give rise to many interesting graph-relational queries in various domains, e.g., finding related proteins retrieved by a relational subquery in a biological network. The maturity of RDBMSs motivated academia and industry to invest efforts in leveraging RDBMSs for graph processing, where efficiency is proven for vital graph queries. However, none of these efforts process graphs natively inside the RDBMS, which is particularly challenging due to the impedance mismatch between the relational and the graph models. In this paper, we propose to manage graphs as first-class citizens inside the relational engine. We realize our approach inside *VoltDB* [6], an open-source in-memory relational database, and name this realization GRFusion. The SQL and query engine of GRFusion are empowered to declaratively define graphs and execute cross-data-model query plans acting on graphs and relations, resulting in up to four orders-of-magnitude in query-time speedup w.r.t. state-of-the-art approaches.

1 INTRODUCTION

Graphs are ubiquitous in various application domains, e.g., social networks, road networks, biological networks, and communication networks [3, 8, 9, 12]. The data of these applications can be viewed as graphs, where the vertexes and the edges have relational attributes [46], or as traditional relational data with latent graph structures [51]. Applications would issue queries that consult the topology of the graphs along with the data associated with the vertexes and the edges or other data sources (e.g., relational tables in an RDBMS). For instance, a user may be interested to find the shortest path over a road network while restricting the search to certain types of roads, e.g., avoiding toll roads.

In an RDBMS, the filtering predicates can be expressed as relational predicates, and they may reference relational tables that have indirect relation with the queried graphs. We refer to these queries as graph-relational queries (or *G+R* queries, for short). *G+R* queries have two main ingredients: 1) graph operations, e.g., shortest-path computation, and 2) relational predicates or relational sub-queries. For example, selecting specific users from relational tables to find the nearest hospitals using shortest-path evaluation on top of a road-network.

As RDBMSs are pervasive and mature, various approaches for using an RDBMS to manage graph data have been proposed, e.g.,

Grail [25] and Aster [45]. The literature has two main approaches that share the idea of building an application on top of an RDBMS to support graphs without modifying the internals of the RDBMS. We refer to these approaches as *Native Relational-Core* and *Native Graph-Core*. In this paper, we propose and investigate a hybrid approach that we term *Native G+R Core* that exploits the strengths of the former two approaches, and we realize our approach inside VoltDB [7, 10].

The *Native Relational-Core* approach (e.g., as in SQLGraph [46] and Grail [25]) embeds a graph inside of relational tables of specific schema. Then, an application on top of the RDBMS is built to translate specific types of graph queries into SQL statements for the RDBMS to execute. For example, Grail can translate shortest-path queries to procedural SQL [25], while SQLGraph translates Gremlin queries with some restrictions [5] into SQL queries [46]. Figure 1(a) illustrates the general architecture of the *Native Relational-Core* approach. Although many graph queries and algorithms are hard to translate into SQL statements, tools can be developed to automate the translation. However, the main issue of the *Native Relational-Core* approach is that the graph operations are evaluated by a sequence of relational operations (e.g., self-joins) that may be more expensive than traversing a native graph representation. Moreover, the *Native Relational-Core* approach does not guarantee an easy-to-comprehend relational schema of the embedded graphs in an RDBMS, e.g., the storage-optimized relational schema generated automatically by SQLGraph is hard for users to understand and write ad-hoc graph-relational queries [46].

The second approach, namely *Native Graph-Core* (e.g., as in Ringo [38], GraphGen [51, 52]), assumes that graphs are already stored in an RDBMS, where an application on top of the RDBMS is built to extract these graphs to analyze them outside the realm of the RDBMS. This approach follows the same philosophy as that of specialized graph databases, where an RDBMS has nothing to do with query execution. Figure 1(b) illustrates the general architecture of the *Native Graph-Core* approach. Notice that a graph in the *Native Graph-Core* requires re-extraction if the relational tables storing the graph in the RDBMS are updated. Moreover, users cannot issue declarative graph-relational queries that reference both the extracted graphs and any other relational data in the RDBMS. One solution to allow graph-relational queries in the *Native Graph-Core* approach is to build another layer that queries both the RDBMS and the extracted graph. This solution is similar to that of Teradata Aster [45], where a data movement fabric and two different query executors (i.e., a relational executor and a graph executor) are used in processing graph-relational queries. However, integrating the results from the graph and the relational executors imposes additional overhead. In summary, the

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

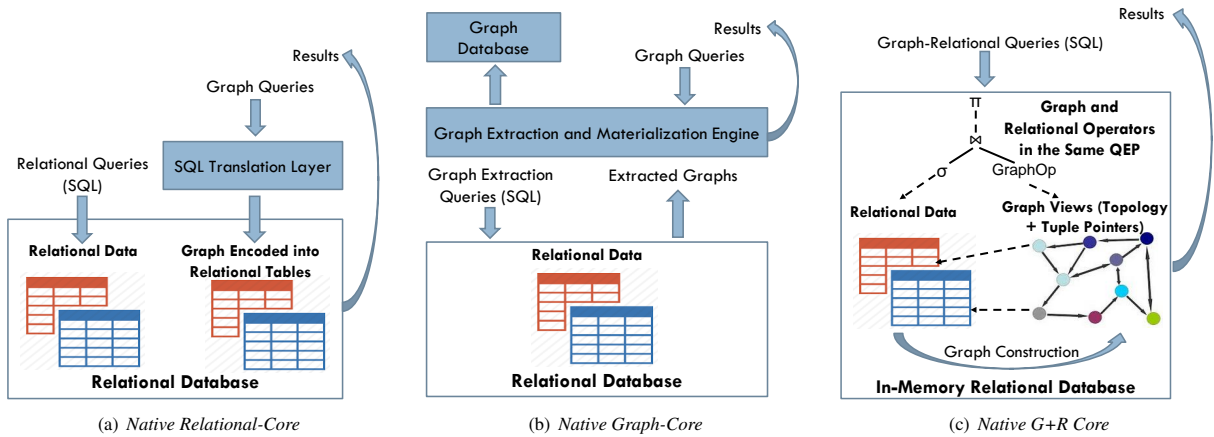


Figure 1: Various approaches for leveraging relational databases in support of graph processing.

	<i>Native Relational-Core</i>	<i>Native Graph-Core</i>	<i>Native G+R Core</i>
Hybrid QEPs	X	X	✓
Native Graph Processing	X	✓	✓
No Query-Translation Overhead	X	✓	✓
No Graph Reconstruction/Re-embedding on Updates	X	X	✓

Table 1: Contrasting various approaches for graph support in RDBMSs.

Native Relational-Core and the *Native Graph-Core* approaches use a vanilla RDBMS, where graphs are not natively recognized by the RDBMS. However, if the necessary layers of the RDBMS are modified to manage graphs as first-class citizens, processing and managing graphs will be more efficient.

In this paper, we investigate a third approach, namely *Native G+R Core*, where graphs are recognized as first-class citizens inside an RDBMS. We address the impedance mismatch between the graph and the relational model, and we realize the *Native G+R Core* approach in a centralized version of VoltDB [7, 10], the open-source implementation of the H-Store in-memory relational DBMS [32]. In-memory data management witnessed early academic and industrial contributions, where the current affordability of large main-memory hardware motivated several and diverse research efforts [14, 15, 23, 30, 33, 35, 36, 38, 39, 44, 49–51, 53].

We refer to our realization of this approach as GRFusion. The main idea of GRFusion is to natively process graphs inside an RDBMS by combining the *Native Relational-Core* and the *Native Graph-Core* approaches under the same umbrella. GRFusion realizes this idea by separating the graph topology from the relational data associated with the vertexes and the edges, and by proposing graph operators to process the graph topology inside the RDBMS, where the graph operators seamlessly co-exist with other relational operators in the same query execution pipeline (or QEP, for short). A graph topology in GRFusion is realized as a native graph structure, where each vertex or edge has pointers to the relational tuples describing their attributes. Hence, a graph topology in GRFusion can be viewed as a traversal index of the relational tuples of the vertexes and the edges. In short, GRFusion presents cross-data-model QEPs, where the inputs to the QEPs can be either relational data or native graph structures.

Figure 1(c) illustrates the general idea of the *Native G+R Core* approach. First, the end-user provides a declarative statement to create graph views that are initialized from relational data, where a graph view is materialized as a new database object. Second, the user is allowed to query the graph views as well as other relational tables or views in the same query. Table 1 contrasts the *Native Relational-Core*, *Native Graph-Core*, and *Native G+R*

Core approaches. The objective of this paper is not to replace the specialized graph systems. However, the main objective is to empower the pervasive relational databases to support graph traversal queries natively and efficiently. Consequently, the relational-data owners can process important class of graph queries through their RDBMS systems without the cost and the overhead of migrating their data and manage it in a separate graph system. The contributions of this paper are as follows:

- Introducing graphs as native objects inside a relational database system, namely VoltDB (Section 3), where online graph updates are supported (Section 3.3).
- Allowing users to seamlessly query and operate on graphs and relations simultaneously and declaratively without leaving the realm of the relational database system (Section 4).
- Introducing graph operators for graph traversals (Section 5.1), and showing their ability to seamlessly co-exist with the relational operators to construct cross-data-model query execution pipelines (Section 5.2).
- Addressing the impedance mismatch between the graph model and the relational model (Section 5.3).
- Conducting an extensive performance study of GRFusion w.r.t. state-of-the-art systems, and reasoning about the benefits of processing graphs in a graph-native representation inside an RDBMS. We compare to SQLGraph, Grail, Neo4j, and Titan, where GRFusion achieves up to four orders-of-magnitude query-time speedup (Section 7).

2 OVERVIEW OF GRFUSION

In GRFusion, graphs are assumed to be initially stored in relations. In the simplest case, a relational table may have a row for each vertex, and another table may have a row for each edge. Also, the vertexes or the edges data can be obtained through a relational materialized view that joins or filters multiple relational tables. To allow flexibility, GRFusion provides the user with a declarative language to define and query graphs (see Figure 2). A graph is defined in GRFusion by what we term *graph views*. A *graph view* identifies the relational tables or the relational views that store

the attributes of the vertexes and edges, namely, the *vertexes relational-source*, and the *edges relational-source*, respectively. *Graph views* define a view of the relational data in the graph model and materializes the graph topology in main-memory in native graph data structures. The materialized graph topology has a *native graph representation* that holds pointers (e.g., tuple identifiers) to the relational data that describe the vertexes and the edges. The main idea behind materializing the graph topology is to empower the relational database engine with the ability to realize complex graph algorithms. Thus, GRFusion helps fill the gap between the relational model and the massive body of research that assumes a graph model. Listing 1 shows how a graph view is created in GRFusion from the relational sources of Figure 3, which is detailed in Section 3.1.

Once a graph view is defined, GRFusion allows the user to write pure graph queries, pure relational queries, or queries that mix both graph and relational operations. GRFusion’s query engine views the relational data in either the relational or the graph model according to the incoming query. In particular, the graph clauses in a query are mapped to graph operators in the QEP, where a graph operator accepts only graph representations as input. GRFusion allows the graph operators and the relational operators to co-exist in the same QEP, where the operator type determines the data model of viewing the data (i.e., graph views for the graph model, and relations for the relational model).

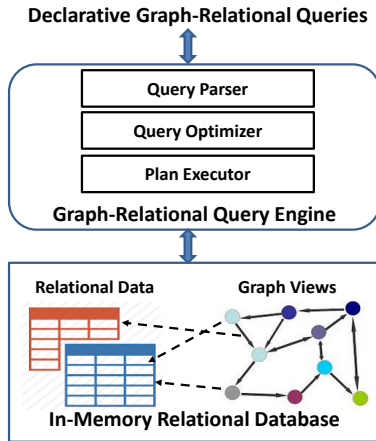


Figure 2: GRFusion’s architecture allows the query engine to process data in both the relational and the graph models.

3 GRAPHS AS DATABASE OBJECTS

As users can create tables in relational databases, they can also create materialized graph views in GRFusion as database objects. A graph view is created once as a singleton object, and can be referenced by multiple users and queries. In Section 3.1, we highlight how graph views are defined declaratively in GRFusion. Section 3.2 illustrates how the topology of a graph in GRFusion is decoupled from the graph data, and how they can be inter-linked. Because dynamic graphs are essential in many applications, the support for graph updates is addressed in Section 3.3.

3.1 Creating Graph Views

GRFusion has a declarative *Create Graph View* statement to create graph views initialized from relational data. The statement has four main objectives: (1) Identifying the name of the graph view to create, (2) Identifying and extracting the graph’s set of vertexes

Users			
uid	fName	lName	dob
1	Edy	Smith	09-25-1971
2	Jones	Parker	11-21-1980
3	Bill	Patrick	02-01-1976
.....

Relationships				
relId	uid1	uid2	startDate	isRelative
1	1	3	01-10-2009	true
2	2	3	12-31-2008	false
.....

Figure 3: A sample social-network in the relational model.

from the underlying relational sources, (3) Identifying and extracting the graph’s set of edges from the underlying relational sources, and (4) Materializing a native graph data structure in memory that reflects the graph topology based on adjacency-list structures. Notice that graph traversal operations can be performed efficiently over this native graph representation and is linked back to the corresponding relational data tuples that describe it. Notice further that the relational source can either be a table or a materialized relational-view because the graph data attributes for the edges and/or the vertexes can be constructed from multiple data sources.

Figure 3 illustrates how a graph view is created in GRFusion. Assume that the data of a social network is stored in the relational tables as in the figure. Tables *Users* and *Relationships* represent the vertexes and the edges of the social network, respectively. Each vertex or edge has an identifier in the relational tables. To illustrate, consider Listing 1 that shows an example of creating a graph view, namely the *SocialNetwork* graph view, in GRFusion from the relational sources in Figure 3. A vertex in the *SocialNetwork* graph has its Id from *Users.uid* and has the two attributes *lName* and *birthdate* that get their values from *Users.lName* and *Users.dob*, respectively. Similarly, Table *Relationships* defines the edges of the *SocialNetwork* graph, where the edge Id comes from *Relationships.relId*, the endpoints come from *Relationships.uid1* and *Relationships.uid2*, and the two edge attributes *sDate*, *relative* refer to Attributes *startDate*, *isRelative* of Table *Relationships*, respectively. For the graph view defined by the *Create Graph View* statement, if the set of vertexes is *V*, and the set of edges is *E*, then, the endpoints of an edge in *E* are constrained to be included in *V*.

Listing 1: A Social Network Graph View Example

```
CREATE UNDIRECTED GRAPH VIEW SocialNetwork
VERTEXES (ID = uid, lstName = lName,
  ↪ birthdate = dob) FROM Users
EDGES (ID = relId, FROM = uid1, TO = uid2,
  ↪ sDate = startDate, relative =
  ↪ isRelative) FROM Relationships
```

3.2 Decoupling the Graph Topology and the Graph Data

The *Create Graph View* statement updates the system catalog of GRFusion to store the definition of the graph view. Creating a graph view results in the materialization of the graph topology as a native graph structure in the main-memory managed by GRFusion (as a singleton object that multiple users and queries can reference). However, the attributes of the vertexes and the edges stored in the relational sources are not replicated in the native graph structure, and main-memory tuple pointers are used to link

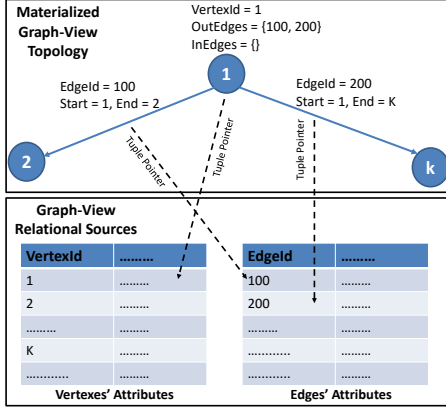


Figure 4: A graph view materializes the topology and holds pointers to the relational data of the vertexes and the edges.

the graph topology to the relational sources. To illustrate, Figure 4 demonstrates how the graph topology is separated from the graph data (i.e., the relational attributes of the vertexes and the edges). As in Figure 4, each vertex or edge has a main-memory tuple pointer that points to the corresponding relational tuple storing the attributes of this vertex or edge. Notice that the design of GRFusion allows a vertex or edge in a graph topology to store multiple tuple-pointers if the relational sources are vertically partitioned (e.g., to support semistructured RDF data, where not all the vertexes or edges share the same set of attributes). Without loss of generality, we assume a single tuple pointer per vertex or edge as the focus is to explore the benefits of empowering an RDBMS with native graph-processing.

The graph topology follows the graph model, where the topology is represented physically as a graph data-structure based on adjacency-lists. The key idea behind this native graph representation is to allow for the efficient execution of graph traversals, where relational joins can be mitigated when traversing a graph. The reason is that materializing the topology of a graph view can be thought of as a traversal index, where each vertex, say V , is associated with the identifiers of both the outgoing edges and the incoming edges of V . Given a graph view, say GV , its topology can be constructed using a single pass over the relational sources defining the vertexes and the edges of GV .

Notice that there is a bi-directional linkage between the graph topology and the graph's corresponding relational data. To illustrate, let T be a relational tuple containing the attribute values of Vertex V . Using the `VertexId` attribute of T , GRFusion can locate Vertex V in the graph representation in $O(1)$ time using the hash map of the native graph structure. Also, using the tuple pointer associated with Vertex V in the graph data-structure, Tuple T can be located in $O(1)$ time. The benefit of separating the graph topology from the graph data is two-fold. First, the size of the graph view is not affected by the size of the graph data that can be very large in some cases. Second, the attributes of the vertexes and the edges in the relational sources can be easily updated without affecting the native graph representation.

3.3 Graph Updates

GRFusion supports *serializable* graph updates that affect the topology or the attributes stored in the relational sources. The topology is affected only when vertexes/edges are added or deleted. GRFusion relies on the design and the implementation of VoltDB to maintain pointers to the relational tuples on memory reallocations.

3.3.1 Graph-Data Updates. Updating the attribute data of an edge or vertex is straightforward as the attributes are stored in relations outside the native graph representation. Hence, these relational attributes can be updated directly. However, updating the `VertexId` and the `EdgId` attributes need special handling because these attributes are used for navigating from the relational store to the native graph structure (e.g., to probe path-traversal operators in a QEP as in Section 5). Although updating the identifiers are not common, GRFusion maintains the consistency of the identifiers in the graph representation when updating their corresponding attributes in the relational sources. Also, GRFusion maintains the referential integrity of the *edges relational-source* when updating a vertex identifier in the *vertexes relational-source*.

3.3.2 Graph-Topology Updates. GRFusion allows topological updates when the relational sources are either relational tables or a relational views selecting from a single table. GRFusion associates each relational source, say R , with the identifiers of the graph views that reference R . When inserting a new tuple into R , the transaction of the insertion statement updates the graph-view topology as part of the transaction (i.e., adding a new vertex or adding a new edge in the graph representation). Similarly, when deleting a vertex or edge, the deletion statement detects the graph views associated with R and updates the affected graph views accordingly as part of the deletion transaction. For example, if R is an *edges relational-source* for a graph view, say GV , the edge in GV corresponding to a deleted tuple is removed from GV .

4 THE PATHS QUERY CONSTRUCT

As graph traversal queries form a massive body of graph queries (e.g., reachability and shortest path queries [19, 24, 26, 42, 43, 47]), GRFusion extends the SQL language to declaratively find paths in graph views. GRFusion introduces the *PATHS* construct to query its graph views. For a graph view, say GV , GRFusion recognizes $GV.PATHS$ in the From clause of a select statement (as it is treated conceptually as a set of paths). Conceptually, this allows GRFusion to traverse and retrieve simple paths from GV that satisfy a path criteria (e.g., predicates on the attributes of the edges forming the path). In addition to $GV.PATHS$, GRFusion recognizes $GV.VERTEXES$, and $GV.EDGES$, to reference the vertexes, and the edges of GV , respectively. We focus on the $GV.PATHS$ construct as the other constructs are straightforward.

GRFusion models a path as an ordered list of edges, where each edge has a start and end vertexes. The edges and the vertexes of a path, say PS , can be indexed and referenced by relational predicates as follows:

- **PS.Edges[StartIndex..EndIndex].EdgeAttribute:** References an attribute of the edges starting from $StartIndex$ until $EndIndex$. A value of '*' for the $EndVertex$ placeholder indicates that all the edges starting from $StartIndex$ should satisfy the relational predicate.
- **PS.Vertexes[StartIndex..EndIndex].VertexAttribute:** References an attribute of the vertexes starting from $StartIndex$ until $EndIndex$. A value of '*' for the $EndVertex$ placeholder indicates that all the vertexes starting from $StartIndex$ should satisfy the relational predicate.

Observe that the aforementioned *EdgeAttribute*, and the *VertexAttribute* placeholders can refer to any attribute of the edges or the vertexes that have been defined at the time of creating Graph-view GV . In addition, each vertex in Path PS has two additional integral attributes, namely *FanIn* and *FanOut*. Also, Path PS allows

accessing to some path-specific properties, e.g., $PS.StartVertexId$ and $PS.Length$ refer to the identifier of the start vertex and the length of Path PS , respectively.

To illustrate how paths can be queried in GRFusion, consider Query Q_p in Listing 2. The From clause of Q_p specifies that the paths are being traversed from the SocialNetwork graph view, where the *vertexes relational-source* of the SocialNetwork graph is Relation *Users*. The query displays the last names of the friends of friends of all the users with Job = ‘Lawyer’. Conceptually, Q_p is evaluated by selecting the sub-graph, say G_{sub} , containing edges with start dates after ‘1/1/2000’. Using Sub-graph G_{sub} , GRFusion explores paths consisting of two edges that originate from the vertexes corresponding to lawyers in the social network. Notice that Listing 2 could use *SocialNetwork.VERTEXES* instead of *Users*. However, Listing 2 uses the *Users* relation to show how relational tables can be joined with the paths of a graph view. Notice that the details of the extended query language of GRFusion are not the main focus of this paper. However, we provide sample code snippets that are relevant to illustrating the evaluation of the graph-relational queries supported by GRFusion.

Listing 2: Friends-of-Friends Path Query Q_p

```
SELECT PS.EndVertex.lstName
FROM Users U, SocialNetwork.Paths PS
WHERE U.Job = 'Lawyer' AND PS.StartVertex.
  ↳ Id = U.uId AND PS.Length = 2 AND PS.
  ↳ Edges[0..*].StartDate > '1/1/2000'
```

Listing 3 presents a reachability query Q_r that queries a protein-interaction network represented by the BioNetwork graph view, and checks if *Protein X* interacts directly (i.e., by an edge) or indirectly (i.e., by a path) with *Protein Y* through either a covalent or stable interaction types. $PS.PathString$ corresponds to the string representation of Path PS . Notice that many paths can exist between the vertexes corresponding to the specified proteins. So, Query Q_r uses the *LIMIT 1* clause because retrieving one path is sufficient to decide on reachability.

Listing 3: Reachability Query Q_r

```
SELECT PS.PathString
FROM Proteins Pr1, Proteins Pr2, BioNetwork
  ↳ .Paths PS
WHERE Pr1.Name = 'Protein X' AND Pr2.Name =
  ↳ 'Protein Y' AND PS.StartVertex.Id =
  ↳ Pr1.Id AND PS.EndVertex.Id = Pr2.Id
  ↳ AND PS.Edges[0..*].Type IN ('
  ↳ covalent', 'stable') LIMIT 1
```

In addition to the ability of referencing the attributes of the edges or vertexes forming a path, say PS , GRFusion allows aggregation functions on the attributes of the vertexes or the edges of PS . The aggregate functions on the attributes of paths have the same usage and constraints as those on relational attributes. For example, if the edges of PS have an attribute, say *Weight*, a query can compute the sum of the weight values across all the edges of PS , i.e., $sum(PS.Edges.Weight)$ can appear in the select-clause of a query to compute the sum of the weights associated with the edges of Path PS .

The PATHS construct can also retrieve sub-graphs based on specific patterns (e.g., the topology of the sub-graph, attributes of the vertexes/edges of the subgraph). For instance, finding triangular structures with specific edge properties, and counting these triangles are important primitives for Machine-Learning,

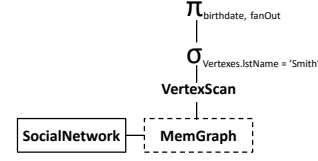


Figure 5: QEP for Query Q_v .

e.g., [48], where a triangle structure can be viewed as a loop of three edges. Listing 4 presents Query Q_t that counts the number of triangles, where the edges have specific values for their *Label* attribute. Notice the use of the *Path.Length* property, where it is necessary to retrieve only triangles (as the sub-graph of interest has only three edges).

Listing 4: Subgraph Pattern Query to Find Triangles Q_t

```
SELECT Count (P)
FROM MLGraph.Paths P Where P.Length = 3 AND
  ↳ P.Edges[0].Label = 'A' AND P.Edges
  ↳ [1].Label = 'B' AND P.Edges[2].Label
  ↳ = 'C' AND P.Edges[2].EndVertex = P.
  ↳ Edges[0].StartVertex
```

More interestingly, paths can be joined to query more complex sub-graph patterns. Similar to relational engines that can perform self-joins for a relational table, GRFusion allows self-joins of the paths of a given graph view. This is possible as the vertexes and the edges of the paths to join can be referenced by relational join predicates.

5 GRAPH-RELATIONAL QUERY PROCESSING

In this section, we explain how GRFusion evaluates graph-relational queries. Section 5.1 introduces the primitive graph operators of GRFusion, while Section 5.2 illustrates how the graph operators integrate with typical relational operators in a cross-data-model QEP, where the graph operators appear in the leaf level of the QEP. Then, Section 5.3 discusses the conceptual query evaluation of graph-relational queries in GRFusion.

5.1 Graph Operators

GRFusion defines three primitive operators to evaluate the graph constructs of graph-relational queries. In particular, GRFusion defines the *VertexScan*, *EdgeScan*, and *PathScan* operators that iterate over a graph view’s vertexes, edges, and paths, respectively. The PathScan operator is a lazy operator following the iterator model [28] to avoid eager generation of paths that might not be required by parent operators. The reason of this design decision is that many queries (e.g. reachability) limit the number of paths to be retrieved, and consequently generating all/multiple paths may be expensive and unnecessary.

5.1.1 Vertex Scan and Edge Scan Operators. Operators *VertexScan* and *EdgeScan* allow GRFusion to iterate over the vertexes and edges of a given graph view, respectively. For example, the *VertexScan* operator provides an alternative access method for accessing the vertexes of a graph view, where the fan-in and fan-out properties of any vertex can be efficiently retrieved in constant time. To illustrate, consider Query Q_v in Listing 5. Q_v selects from the set of vertexes of the SocialNetwork graph view, and then applies some relational operators afterwards. To evaluate Q_v , GRFusion constructs the query execution pipeline, say QEP_v ,

as in Figure 5. Operator *VertexScan* scans the vertexes of the graph defined by the *SocialNetwork* graph view from the in-memory graph structure (represented as *MemGraph* in Figure 5, that references the singleton graph structure of the graph view). Vertexes with last name ‘Smith’ are selected and a relational projection operation selects only the birth date and the fan-out properties.

Listing 5: Vertexes Selection Query

```

SELECT VS.birthdate, VS.fanOut
FROM SocialNetwork.Vertexes VS
WHERE VS.lstName = 'Smith'

```

5.1.2 The PathScan Operator. In GRFusion, the *PathScan* operator is responsible for traversing a graph view to construct simple paths identified by a graph query. *PathScan* is a logical operator that has three physical operators with three corresponding graph-traversal algorithms. All the physical operators explore a traversed vertex only once to avoid loops, i.e., the paths in GRFusion are simple paths. In particular, the query optimizer maps a logical *PathScan* operator into *DFScan*, *BFScan*, or *SPScan*, corresponding to depth-first search, breadth-first search, or shortest-path search physical operators, respectively. In this section, we focus on the logical semantics of the path scan operator. We defer the discussion of the physical operators to Section 6.

As a logical operation, the paths-discovery process in GRFusion starts from a set of start vertexes to avoid materializing all possible paths. These start vertexes are either stated explicitly in the query (e.g., $PS.StartVertex.Id = Value$) or are generated by other operators during query evaluation (e.g., $PS.StartVertex.Id = VS.Id$ as in Listing 2). In the latter scenario, the start vertexes selected by some operators (e.g., *TableScan*, relational sub-query), are used to probe the *PathScan* traversal operator. If the start vertexes of a path selection are not defined, all the vertexes of the corresponding graph view will be used as starting vertexes. Notice that the paths in GRFusion are not eagerly materialized by a *PathScan* operator, rather they are lazily generated.

To illustrate how paths are explored in GRFusion, consider Query Q_p in Listing 2. Q_p explicitly states that the path discovery process starts from the vertexes corresponding to lawyers in the social network. Figure 6 demonstrates the query evaluation pipeline QEP_p that evaluates Query Q_p , where *MemGraph* refers to the singleton materialized graph structure of the graph view. In particular, Q_p starts the traversal process from each qualified vertex. Notice that the qualified vertexes are retrieved using a relational operator (e.g., by a *TableScan* or *IndexScan* operators) in Figure 6. The reason is that using a relational access method with filtering predicates on the *vertexes relational-source* is more efficient than using the tuple pointers in the graph view to filter all the vertexes on the fly. Because of the seamless integration of the relational and graph models in GRFusion, this optimization alternative is feasible. While traversing the graph view, only the edges with start dates after ‘1/1/2000’ are considered. Also, QEP_p explores paths of length two only (i.e., consisting of two edges) that originate from a given start vertex. As an effective optimization, GRFusion pushes predicates, e.g., path-length predicates, to be considered during the traversal process. This optimization allows GRFusion to apply early pruning of paths, and to reduce the size of the intermediate results flowing through the query pipeline. Consequently, the performance of the query evaluation process is boosted w.r.t. the processing time as well as the temporary memory used for the intermediate results.

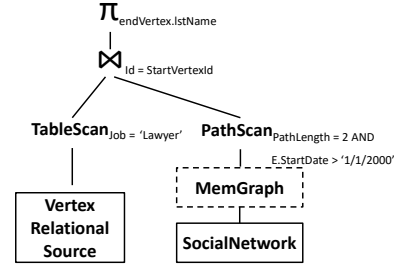


Figure 6: GRFusion joins a relational table with a graph-view traversal-operator for Query Q_p .

5.2 Cross-Model Query-Execution-Pipelines

A query in GRFusion can reference relations or relational views with graph views simultaneously. A pure relational engine has a main structure (i.e., tuple) that is passed among the relational operators in a query evaluation pipeline (QEP). GRFusion allows its query engine to view data by two different data models, namely, the relational model and the graph model. GRFusion allows a single QEP to have two main categories of operators that interact seamlessly in a QEP. The first category contains the relational operators (e.g., select, project, relational join) that can interact directly with relational tables. The second category contains graph operators that can operate on graph views. GRFusion integrates both categories of operators by allowing a relational operator to operate on the result of a graph operator. In particular, GRFusion unifies the interface of the output of both the relational and the graph operators. Specifically, the query engine of GRFusion abstracts graph processing by using three data types that extend the *Tuple* data type, namely the *Vertex*, *Edge*, and *Path* data types, where each has a schema that depends on the queried graph-view, as explained below.

In GRFusion, a vertex, say V , is represented in a QEP by a tuple, say T , where each attribute of V becomes an attribute in T . For example, a graph vertex in Listing 1 is represented by a tuple with attributes: $(uid, lstName, birthdate)$. In addition, Vertex V has the following properties:

- **FanOut:** Contains the number of V ’s outgoing edges.
- **FanIn:** Contains the number of V ’s incident edges.

An edge E is represented by a tuple with attributes corresponding to E ’s attributes in addition to the following attributes:

- **From:** Contains the start vertex of Edge E .
- **To:** Contains the end vertex of Edge E .

GRFusion defines the *Path* data type, where a path, say P , is a sequence of identifiers of the edges that form P . In particular, P is an extended tuple with the following attributes defining its schema:

- **Length:** Is the number of edges in P .
- **StartVertex:** Is the start vertex of P .
- **EndVertex:** Is the end vertex of P .
- **Vertexes:** Is the list of vertexes forming P .
- **Edges:** Is the list of edges forming P .

5.3 Conceptual Evaluation of Graph-Relational Queries in GRFusion

GRFusion addresses the impedance mismatch between the graph model and the relational model by unifying the type of the elements that move among the relational and the graph operators within a QEP. To illustrate, we list below the high-level steps

that describe GRFusion’s conceptual evaluation of declarative graph-relational queries, i.e., ones that reference relation(s) and graph-view(s):

- The relational tables and views are joined together using all the relational predicates in the WHERE clause of the query. This step yields a single resultant relation, say R .
- Each graph operator operates on a graph view, say GV , using its in-memory singleton graph-structure, say Mem_{GV} . In case of using different aliases on the same graph view, each alias is assigned an independent pointer to Mem_{GV} .
- When querying a combination of relations, relational views, vertexes, edges, or paths, all the graph operators operate only on graph views. Observe that the output of each graph operator is an extended type of the relational *Tuple* type. Hence, the output of the graph operators can be ingested by the relational operators (e.g., the joins) in the same QEP seamlessly, where a relational join outer tuple can be used to probe a graph operator in the inner (e.g., see Figure 6).
- The predicates in the WHERE clause of the query that have not been consumed in producing R are used to join R with all the vertexes, edges, and paths referenced by the query.
- The SELECT list is used to perform projection.

6 QUERY OPTIMIZATION

GRFusion optimizes graph-traversal queries with two objectives in mind: (1) pruning undesired paths as early as possible to optimize the runtime, and (2) favoring traversal algorithms with less-memory requirements. The second goal is vital as memory should be consumed discreetly in an in-memory system. Optimization techniques for early pruning are discussed in Sections 6.1 and 6.2. In Section 6.3, we address the traversal-algorithm selection.

6.1 Path Length Inference

The query optimizer of GRFusion infers the allowed length of the paths described by the queries. The main objective is to make sure that a path returned from the PathScan operator is unlikely to be rejected by a parent operator (e.g., a join operator) due to a predicate referencing the path length. For instance, if a query has the filter " $PS.Edges[5..*].Att1 = Value$ ", then PathScan infers that the minimum path length to return is 6 (indexing is zero-based). Hence, PathScan will not return a path of length 5 or less. Many real-world queries specify the length of the desired paths, e.g., triangle-counting queries [48] specify a path length of three, the popular friends-of-friends queries restrict a path length to two, and many reachability queries put a cap on the maximum length of the path connecting the queried endpoints.

For each collection of paths, say PS , that is referenced in the *From*-clause, the query optimizer analyzes the predicates referencing the length of PS explicitly (e.g., $PS.Length = value$), or implicitly (e.g., by analyzing the logical operators as in $PS.Edges[5..*].Att1 = X$ AND $PS.Edges[7..9].Att2 = Y$), to predict the range of allowed lengths of the paths to return. Then, the inferred path length is considered by PathScan while traversing the graph (e.g., an inferred maximum path length of 8 will prune any path of length ≥ 9).

6.2 Pushing Filters Ahead of Path Scans

To prune paths early, all the filters related to discovering the paths of a graph view are pushed ahead of the *PathScan* operator. For instance, for a graph view’s paths, say PS , Predicate " $PS.Edges[0..*].Cost < 10$ " is pushed so that *PathScan*

can prune any potential path explored with an edge of cost ≥ 10 . Similarly, predicates that refer to aggregates on a path’s attributes will be computed and checked during the *PathScan* evaluation. For example, consider a query, say Q , with the predicate " $Sum(PS.Edges.Cost) < 100$ ". When *PathScan* explores Path P while evaluating Q , *PathScan* will accumulate the cost-attribute of the edges of P during the traversal. If the accumulated cost exceeds 100, P will be dropped and will not flow to the operators next in the QEP.

6.3 Logical to Physical Operator Mapping

Recall from Section 5.1.2 that the *PathScan* operator is a logical operator that is mapped into one of three physical traversal operators for execution, namely, depth-first search, breadth-first search, and shortest-path search based on Dijkstra’s algorithm [24].

The shortest-path physical operator, namely *SPScan*, is very useful in top-k shortest path queries. Listing 6 illustrates how the user can instruct the optimizer to use *SPScan*. Given a non-negative numerical edge attribute, *SPScan* traverses the graph using Dijkstra’s algorithm [24], and returns the next shortest-path as requested (i.e., pulled) by the parent operator in the QEP. *SPScan* is useful in many applications, e.g., recommendation systems and route discovery, to avoid the costly straightforward plan, i.e., avoid enumerating all paths, then filtering, sorting, and then returning the top ones.

For general graph-traversals where shortest paths are not defined, GRFusion can use either a depth-first search (i.e., a *DFScan* operator), or a breadth-first search (i.e., a *BFScan* operator). The user can give a query hint to decide on depth-first or breadth-first evaluations. To illustrate how GRFusion decides on the physical operator to perform a general graph traversal in the absence of an explicit query-hint, assume that a query, say Q , searches for Path P of Length L . Assume further that Query Q targets a graph view where the average fan-out is F . Following an analysis similar to that in [41], a depth-first search can contain on average $F * L$ vertexes in its stack data structure. In contrast, a breadth-first search can contain F^L vertexes in its queue data structure. Hence, GRFusion uses BFS if $F < \sqrt[L]{L}$ to optimize for memory. This optimization is applicable if the path length can be inferred and by maintaining the average fan-out statistic for each graph view in the system catalog. Otherwise, GRFusion uses the default scan operator that the user can set based on the expected workload (e.g., BFS can still be better if the underlying graph has a large diameter and frequent queries find the desired paths after one or two hops). GRFusion has a configuration to store the average fan-out of graph views as a statistics object. If this configuration is enabled, GRFusion runs a thread in the backend to compute the average fan-out using the compact graph-view structures.

Listing 6: Declarative Shortest-Path Query

```
SELECT TOP 2 PS
FROM RoadNetwork.Paths PS HINT(SHORTESTPATH
  ↳ (Distance)), RoadNetwork.Vertexes
  ↳ Src, RoadNetwork.Vertexes Dest
WHERE PS.StartVertex.Id = Src.Id AND PS.
  ↳ EndVertex.Id = Dest.Id AND Src.
  ↳ Address = "Address 1" AND Dest.
  ↳ Address = "Address 2"
```

7 EXPERIMENTAL EVALUATION

We experimentally evaluate the performance of GRFusion, a realization of the proposed *Native G+R Core* approach inside a centralized version of VoltDB. We compare GRFusion to the state of the art of the *Native Relational-Core* approach, namely SQLGraph [46], and we compare to Grail [25]. Although Grail uses a different computational model than GRFusion, they both have the common ground of executing queries through an RDBMS. We also compare GRFusion to two popular specialized graph systems, Neo4j [4] and Titan [11]. The reason for comparing with specialized graph systems, which follow the *Native Graph-Core* approach, is to show that graph-traversal queries can be efficiently handled by GRFusion.

Mitigating the disk IO cost from the baselines: As GRFusion is an in-memory system, the experiments are designed to mitigate the disk cost of all the baselines we compare to. We implemented SQLGraph and Grail as described in [46], and [25], respectively, on top of the in-memory VoltDB system. We configured Titan to use the in-memory storage configuration, and we set Neo4j to run and execute over a RAM disk on Linux.

We consider two important categories of graph queries, namely, traversal-based queries and pattern-matching queries, where the queries can take additional filtering predicates. For traversal-based queries, we evaluate reachability queries (e.g., Listing 3). We also evaluate shortest-path queries to compare with Grail [25]. For pattern-matching queries, we evaluate the triangle-counting query using filtering predicates on the edges while varying selectivity. The triangle-counting query is a primitive operator in many machine-learning and knowledge-discovery techniques, e.g., [48]. Experiments are conducted on a machine running Linux kernel 3.17.7 on 32 cores of Intel Xeon 2.90 GHz with 384 GB of main-memory.

7.1 Datasets

We use real graph datasets that represent four different application domains, namely, road networks, biological networks, authorship networks, and social networks. For the road networks, we use the continental-sized Tiger dataset [9] that covers the entire U.S., where the edges represent road segments, and the vertexes represent road intersections. For the biological networks, we use the String protein-interaction dataset [8], where the vertexes represent proteins, and the edges represent interactions among the proteins. We use the DBLP [1] dataset for the authorship networks, where the vertexes represent authors, and the edges represent co-authorship relations. We use the Twitter dataset [3] for the social-network application, where the dataset represents the follower graph of Twitter. The vertexes in Twitter represent users, where an edge from User A to User B denotes that User A follows User B. Table 2 summarizes the properties of these datasets.

Controlling sub-graph selectivity: We study the effect of selecting a subgraph from an underlying graph before performing a graph operation (e.g., selecting a sub-graph containing 10% of the edges of the entire graph before executing a shortest-path query or a topological pattern-matching query on the selected sub-graph). For each dataset, we vary the selectivity of the queries from 5% to 50%.

Evaluating the effect of graph-views in the *Native G+R Core* approach: To accurately study the performance gains due to the graph-views of the *Native G+R Core* w.r.t. the *Native Relational-Core* approach, we use breadth-first search instead of depth-first search, and we do not push the predicates ahead of the path scan operator in GRFusion for all the reachability-queries experiments.

7.2 Unconstrained Reachability Queries

We contrast the performance of GRFusion with that of SQLGraph, Neo4j, and Titan, when processing reachability queries without filtering predicates on the graph edges. Given two nodes, say A and B , a reachability query returns true if a path exists from Node A to Node B . The query-processing time of a reachability query is affected by the path length of the query result. The reason is that the increase in the number of edges traversed directly corresponds to the number of relational joins in the *Native Relational-Core* approach (e.g., SQLGraph).

For each dataset in Table 2, we generate random reachability queries with different path lengths that make the query endpoints connected. We vary the path length from 2 to 20. For each path length, say l , we generate 10,000 random queries, say Q_l . We run Q_l and measure the average query-processing time using GRFusion, SQLGraph, Neo4j, and Titan.

Figure 7 shows the average query-processing time of running the queries using all four systems, where the x-axis and the y-axis give the path-length of the query answers and the query-processing time in milliseconds, respectively. GRFusion achieves up to four orders-of-magnitude speedup in query-processing time compared to SQLGraph, where the speedup increases as the graph size increases. For instance, the speedup reaches 599x for the DBLP graph, and 2483x for the larger String graph. The reason is that GRFusion uses the compact graph view that captures the graph topology, where the graph views act as navigational indexes. Hence, GRFusion does not perform any relational join on the relational sources to traverse the graphs. In contrast, SQLGraph performs a relational join for each edge traversal during the path discovery process. Consequently, the query-processing time in SQLGraph increases as the path length of the query result increases. Moreover, the SQLGraph approach may not scale in main-memory RDBMSs when the graph size is very big due to the size of the intermediate results of the relational joins. To illustrate, in Figure 7(d), in the Twitter dataset, the *Native Relational-Core* represented by SQLGraph does not execute if the query evaluation requires more than four relational joins. The reason is that the intermediate temporary-memory of the join operators exceeds 6 GB, which is 60 times the 100-MB recommended limit in VoltDB. To allow room for query-evaluation pipelining to reduce the intermediate results, and to mitigate the limits of the main-memory, we execute the Twitter queries on a popular disk-based commercial RDBMS. The queries on the Twitter graph time-out after 5 hours of execution when the traversal depth of the queries exceeds four. In contrast, the systems following the *Native Graph-Core* represented by Neo4j and Titan scale for deep graph-traversal queries on large graphs as the overhead of the relational joins does not exist, where a deep graph-traversal query is a query that explores paths of long lengths, i.e., many edges, which corresponds to many joins in the *Native Relational-Core*. However, GRFusion that realizes the proposed *Native G+R Core* approach is able to scale for deep graph-traversal queries with better performance than those of the native graph systems.

Comparing GRFusion to the specialized graph databases Neo4j and Titan, GRFusion has a query-time speedup that exceeds three orders-of-magnitude for the String graph (see Figure 7(c)). We attribute these performance gains of GRFusion over the specialized graph databases to implementation factors and not to a fundamental change in the computational model. The reason is that GRFusion is based on VoltDB that has a low-overhead concurrency model (e.g., no lock overhead as in the specialized graph

Dataset	Number of Vertexes	Number of Edges	Construction Time	Memory Size (GB)
Tiger Road Network	24,412,259	58,698,439	2.08 Min	0.88
DBLP Co-Author Network	1,007,047	6,592,656	1.59 Sec	0.09
String Protein Network	1,520,673	348,473,440	3.81 Min	4.17
Twitter Follower Network	41,652,230	1,468,365,182	10.87 Min	17.81

Table 2: The graph views in GRFusion are fast to construct with low memory overhead for the datasets of the evaluation.

databases). Moreover, VoltDB has an optimized memory manager written in C++ that is significantly more efficient than the JAVA memory managers of both Neo4j and Titan. Theoretically, if we remove all the implementation-specific factors, the performance of GRFusion should be comparable to that of the specialized graph systems as both are processing native graph representations. In Section 7.3, we present the performance of GRFusion when evaluating queries that do not only consult the graph topology, but also the edges’ attributes stored in the relational sources.

7.3 Reachability Queries with Filtering Predicates

We evaluate the performance of reachability queries in GRFusion and compare it to the baselines when the queries are associated with a filtering predicate. To study the effect of sub-graph selectivity (i.e., selecting the sub-graph to perform the query on), we generate reachability queries similar to the ones described in Section 7.2 with varying selectivities. We vary the selectivity parameter from 5% to 50% using synthesized edge attributes to control the selectivity. We limit the path length of the results of the generated queries to 20 to emphasize the effect of the selectivity of the sub-graph to operate on.

Figure 8 shows the average query-processing time for executing the reachability queries with filtering predicates using all 4 systems and datasets, where the x-axis and the y-axis are the edge-selectivity of the queries, and the query-processing time in milliseconds, respectively. Observe that, for the relatively-small DBLP graph in Figure 8(a), SQLGraph outperforms Neo4j and Titan as the relational engine can execute joins and apply filtering predicates efficiently on relations of small cardinalities. GRFusion outperforms both SQLGraph and the specialized graph engines. There are two main reasons behind GRFusion’s performance gains. First, GRFusion uses a compact graph data structure to perform the traversal and avoids relational joins completely to explore the underlying graph. Second, GRFusion relies on the relational engine to evaluate the filtering predicates on the edges. Recall that GRFusion has a direct pointer to an edge’s tuple that is accessed in $O(1)$ time to evaluate the query filtering-predicate using the efficient logic of the relational engine. Hence, GRFusion combines the strengths of both the graph systems and the relational systems to achieve the best-of-both-worlds in terms of performance. However, the efficient evaluation of the filtering predicates and the cost of the relational joins in SQLGraph do not pay off when the size of the relations increase. To illustrate, refer to Figure 8(b), where the performance of SQLGraph degrades as more edges are selected. For the String dataset in Figure 8(c), SQLGraph exceeds the temporary memory limits of VoltDB after selecting a subgraph of size larger than 25% of the queried graph for the reasons illustrated in Section 7.2. For the largest Twitter dataset, SQLGraph is not able to perform even on a subgraph of a 5% selectivity. The reason is that the cost of 20 relational joins on the large Twitter table exceeds the temporary-memory limits of VoltDB, and time-out

the queries on a commercial disk-based RDBMS after 5 hours of execution. Also, as the number of self-joins increases in the *Native Relational-Core* approach, the relational optimizer may not be able to select the best join algorithm due to inaccurate cardinality estimations of the intermediate results (see [27] for details).

The relational engine is efficient in performing filtering predicates. This set of experiments demonstrates the power of extending the relational engine with a native graph-core processor that is optimized for graph traversals and that uses efficient memory representation. Figure 8 demonstrates the scalability and the efficiency of GRFusion in contrast to the baselines in handling graph queries with filtering predicates. Notice that increasing the edge-selectivity factor of the queries has less impact on Neo4j, Titan, and GRFusion than on SQLGraph w.r.t. query-processing time. The reason is that these queries are evaluated on a graph structure by performing the filtering predicates on the fly as the graph is being traversed. The selectivity affects the query performance of all the approaches. However, it is more impactful in the case of pure-relational evaluation. For example, in Figure 8(b), the processing time of SQLGraph increases by 138x when changing the selectivity from 5% to 50%, in contrast to an increase of 1.72x in GRFusion on the same setup.

7.4 Sub-Graph Pattern Matching

We evaluate the performance of the triangle-counting query. Given a graph, say G , a triangle-counting query, say Q_{TC} , counts all the sub-graphs of a triangle pattern (e.g, see Listing 4). Notice that the *Native Relational-Core* approach, e.g., SQLGraph, can scale for this specific pattern query as only two relational joins are needed for query evaluation. This is the reason for choosing this pattern query besides its importance as a primitive in many applications [48]. Figure 9 gives the performance of evaluating triangles queries on the DBLP, Tiger, and String graph datasets, where the x-axis and the y-axis are the edge-selectivity of the queries and the query-processing time in milliseconds, respectively.

Notice that in Figure 9, the SQLGraph approach outperforms both Neo4j and Titan when the selected sub-graph size is small, e.g., up to a selectivity of 10% for the DBLP dataset as in Figure 9(a). Also, notice that SQLGraph is more sensitive to the selectivity parameter than all the other approaches including GRFusion. Although only two joins are required by SQLGraph in this type of queries, increasing the number of tuples to join increases the query processing time, which results in better performance by Neo4j and Titan when increasing the selectivity parameter. For instance, Neo4j and Titan are more efficient than SQLGraph for the String dataset in Figure 9(c) for a selectivity parameter greater than 20%.

Figure 9 illustrates that GRFusion outperforms SQLGraph, Neo4j, and Titan by up to one order of magnitude in query performance. We attribute this performance advantage by GRFusion to the same reasons reported in Section 7.2.

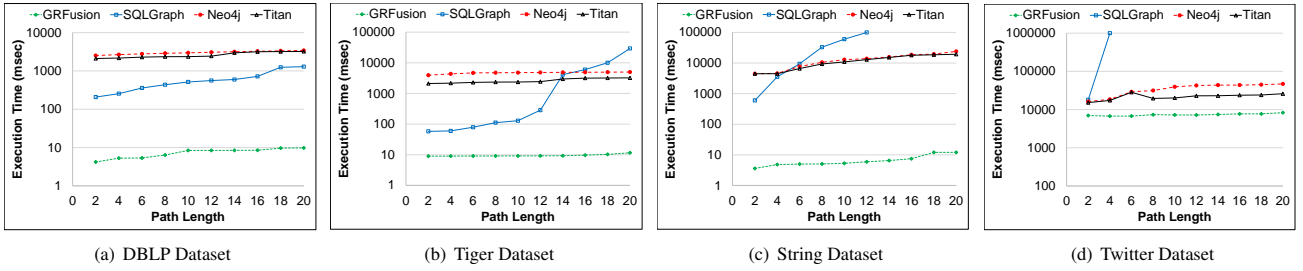


Figure 7: GRFusion achieves up to 4 orders-of-magnitude query-time speedup for *unconstrained reachability queries*.

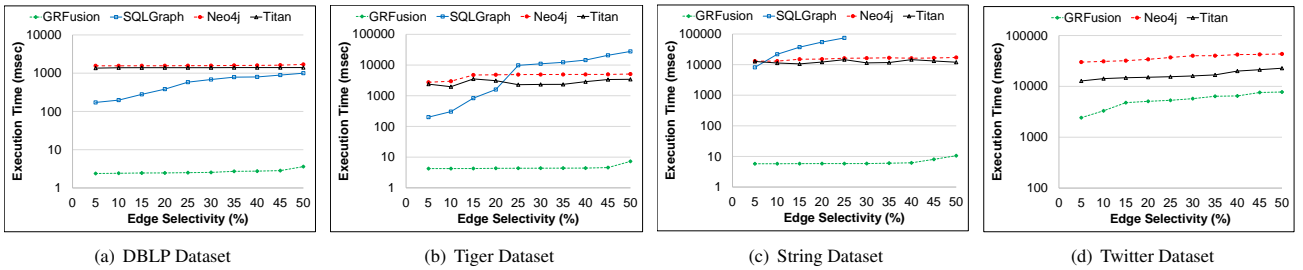


Figure 8: GRFusion achieves up to 4 orders-of-magnitude query-time speedup for *reachability queries with filtering predicates*.

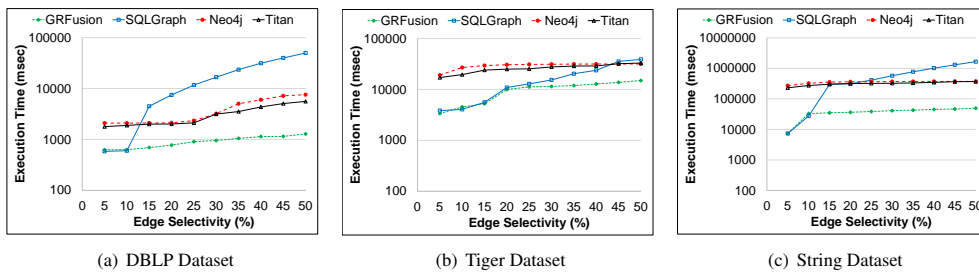


Figure 9: GRFusion finds all the triangles with filtering predicates with a query-time speedup of one order-of-magnitude.

7.5 Shortest-Path Queries with Filtering Predicates

We conduct an experiment using the Tiger road network to assess the performance of GRFusion in evaluating the single-source shortest-path query (or SSSP, for short) in contrast to Grail [25]. The purpose of this experiment is to show that a simple algorithm, e.g., Dijkstra’s algorithm [24], executing inside a relational database system can achieve significant performance gains over a pure-relational approach, e.g., as in Grail [25], when evaluating SSSP queries, or more generally, intensive traversal queries. Notice that the computational model of Grail is based on the vertex-centric computational approach that is different from the graph-traversal model of GRFusion. However, both approaches have a common ground due to using an RDBMS in the evaluation. We implement the SSSP query of Grail as reported in Listing 3 in Grail’s paper [25]. Our Grail implementation is an in-memory implementation on top of VoltDB to mitigate the disk IO cost, and we allow Grail to filter the edges while processing to report the effect of sub-graph selections on the query-execution performance.

We generate 1000 random sources from which we execute an SSSP query to all the other vertexes, and we report the average query execution time for various sub-graph selectivity factors.

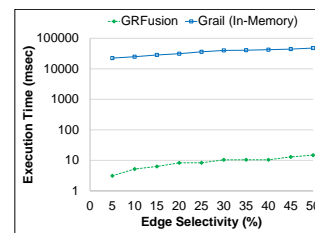


Figure 10: GRFusion executes SSSP queries natively inside an RDBMS few-thousand times faster than Grail.

Figure 10 gives the performance of evaluating SSSP queries on the Tiger road network, where the x-axis and the y-axis are the edge-selectivity of the queries and the query-processing time in milliseconds, respectively. GRFusion achieves more than three orders-of-magnitude query-time speedup w.r.t. Grail. Notice that we do not use an advanced SSSP evaluation method. Instead, we use a straightforward Dijkstra’s algorithm that utilizes efficient filtering-predicates of the relational database engine. This emphasizes the point that having a native and an efficient graph representation inside an RDBMS can fill the gap between the RDBMSs and

the graph algorithms that are designed for native graph structures, where these graph algorithms can achieve significant performance gains when compared to equivalent pure-relational query evaluation approaches.

7.6 The Overhead of Graph Views

As graph views are materialized in GRFusion, we report the construction time as well as the consumed memory space for each dataset. Table 2 illustrates that the construction time ranges from two seconds to 10 minutes according to the size of the graph. The reason is that the construction process passes only once by the vertexes relational-source as well as the edges relational-source. Similarly, Table 2 shows the memory size due to the materialization of the topology of every graph. The consumed memory is of acceptable overhead because only the graph topology is materialized, where each vertex and edge holds pointers to the relational data instead of replicating the relational data inside the graph views. For example, only 0.88 GB is needed to construct a graph view for the continental-sized US road network. Moreover, the overhead of updating the graph views is low. On average, it takes 0.04 milliseconds to add a new edge into an existing graph view, i.e., the total time to insert a tuple in the relational source as well as updating the topology of the corresponding graph view. For both the deletions and insertions of vertexes and edges, GRFusion incurs 5%-11% additional overhead to the time of manipulating the relational sources. The reason for this low overhead is that the logic of manipulating the graph views is linear in time w.r.t. the number of affected vertexes or edges as illustrated in Section 3.3.

8 RELATED WORK

Graphs Integration with Relational Databases: There is a plethora of database systems that adopt the graph data model (e.g., Neo4j [4] and Titan [11]). These systems have powerful graph querying features. However, it has been shown that for many graph queries, the performance of these systems can be achieved or exceeded by a vanilla relational database [25, 46]. For graph-relational queries, a graph database is useful if it is feasible to: a) import all the relational data into the graph database, or b) develop a custom layer where results from the graph database and the relational database are integrated to form the final results. In contrast, GRFusion allows efficient execution of graph-relational queries with neither the overhead of importing data nor the overhead of integrating query results from different systems. Commercial systems, e.g., Oracle Graph and Aster [45], follow the architecture of processing graph-relational queries using different run-time systems, where the results are combined at the end. For example, Aster allows defining graph functions that can be referenced in the FROM-Clause of a SQL statement. During query execution, the graph function is extracted and evaluated using a graph runtime system. Eventually, the result from the **external** graph-runtime is transformed into a relational table that can be integrated with the parent SQL query. Similarly, G-SPARQL [40] is a SPARQL-like language for querying attributed graphs, where a graph is represented and processed using a hybrid Memory/Disk model, and the query-execution is split between the RDBMS and a memory-based layer outside the RDBMS. In contrast, GRFusion executes the graph operations as well as the relational operations of a query through a cross-data-model QEP without leaving the realm of the RDBMS.

Several graph libraries and systems target graph analytics, e.g., CRAY Graph Engine [13], Pregel [34] and its open source version

Giraph [2], GraphLab, GraphFrames [22]. For graph analytics, it may be acceptable to import data from relational databases for analytical purposes. In contrast, GRFusion also serves OLTP scenarios. This is possible as the relational data in GRFusion is not deeply copied into the graph views. Moreover, the updates to the relational data that affect the topology of the defined graph views incur little overhead to update the in-memory graph structures in GRFusion.

Relational Databases with Modified Layers for Graph Processing: In this category, the internals of an RDBMS are modified to some extent, but not to a level that executes a graph-relational query through the same QEP as in GRFusion. For example, SAP HANA Graph and GRAPHITE [37] allow graph operations to directly execute on the relational data in a column-store without replication. However, two different runtime components execute the graph-relational queries. In contrast, GRFusion uses a single runtime leading to better performance. In [18], an access method is proposed to process graphs stored on disk under certain locality assumptions. In contrast, GRFusion is a main-memory system that traverses a graph by realizing a light-weight structure describing the graph topology.

Extracting Graphs from Relational Databases: In this category, graphs stored in relational tables are extracted from the database system to be under the control of an independent application. This independent application allows for querying the extracted graphs using graph APIs. Ringo [38] and GraphGen[51, 52] are representatives of this approach. In contrast, GRFusion processes graphs inside the relational database and does not extract the graphs outside the realm of the database engine. Additionally, GRFusion supports dynamic graphs, where online updates are possible. Notice that to support graph-relational queries, e.g., in Ringo or GraphGen, the relational part of the query should be processed by the relational database, and the graph operations should be processed by Ringo or GraphGen, where another external layer will be responsible for integrating the graph results and the relational results into the final query result.

Encoding Graphs in Relational Databases: In this line of work (e.g., SQLGraph [46], Grail [25]), graphs are stored in relational tables with schema optimized for specific graph queries. After encoding graphs in a vanilla relational database, a translation layer is designed to translate the supported graph queries into complex SQL statements for the relational database to execute. Although the query performance of this approach is comparable to specialized graph databases for specific queries, these systems make it difficult for users to write declarative graph-relational queries. In particular, the schema of the relations storing the graph data may not be suitable for users to query directly and join with other relational data. The reason is that the schema is usually auto-generated based on the input graph for optimization purposes.

Tailored Operators for Specific Graph Operations: In this category, several research efforts (e.g., [17, 20, 21]) have been conducted since the 1980s and until recently (e.g., [16, 26]). However, most of these efforts target specific query types (e.g., transitive closure, shortest paths). Unlike GRFusion, these approaches also do not support a unified/cross-model declarative language to query both graph and relational objects simultaneously. In [17, 20], Relational Algebra is extended with operators to allow for recursive queries. Although the proposed recursive algebra helps execute some graph traversal queries, query execution is not efficient because the graph operators execute over relational tables and not over native graph representations. For instance, several iterations with insertions into temporary tables are needed to keep track

of the traversal state. Similarly, Vertica [31] presents optimizations for graph-relational queries. However, the graph operations execute over pure relational structures and not on graph representations. Thus, costly relational joins are mandatory in many cases to traverse graphs. In contrast, GRFusion’s graph operators process native graph structures in main-memory without performing costly joins and without manipulating temporary tables to traverse a graph topology. Dar et al. [21] use relational operators repetitively to compute the transitive closure of a graph represented in a predefined relational schema. Gao et al. [26] present specific optimizations to process shortest-path queries over graphs stored in a relational database. GRFusion is more general and can join graph views with relational tables in the same query. Moreover, GRFusion addresses the impedance mismatch between the graph model and the relational model. In EmptyHeaded [16], graphs in a relational storage are queried using a datalog-like language [29]. The core idea of EmptyHeaded is to leverage join algorithms with strong theoretical guarantees in addition to using advanced query-compilation techniques. In contrast, GRFusion avoids relational joins completely when traversing the topology of a graph view.

9 CONCLUSION

We introduce the notions of in-memory materialized graph views, graph operators that seamlessly integrate with relational operators in query evaluation pipelines, memory management, and query optimization techniques for optimizing graph-relational queries. GRFusion is a realization of the proposed *Native G+R Core* approach inside VoltDB. The key idea behind GRFusion is to show the effect of extending an RDBMS to handle natively and seamlessly graph and relational data through cross-data-model QEPs. We introduce the PATH construct, and the extended SQL language of GRFusion to declaratively express graph-relational queries. GRFusion constructs in-memory graph structures to capture the graph topology and exploits the relational engine’s power in evaluating the relational constructs of the queries. Consequently, GRFusion efficiently handles deep graph-traversal queries without any relational joins to explore the connectives of the vertexes of a graph. We evaluate GRFusion using various graph queries w.r.t specialized graph engines and systems following the *Native Relational-Core* approach, where GRFusion achieves up to four orders-of-magnitude query-time speedup.

REFERENCES

- [1] <http://dblp.uni-trier.de/xml/>.
- [2] <http://giraph.apache.org/>.
- [3] <http://konect.uni-koblenz.de/networks/twitter>.
- [4] <http://neo4j.com/>.
- [5] <https://github.com/tinkerpop/gremlin/wiki>.
- [6] <https://github.com/voltdb/voltdb/>.
- [7] <https://github.com/voltdb/voltdb/>.
- [8] <http://string-db.org/>.
- [9] <https://www.census.gov/geo/maps-data/data/tiger.html>.
- [10] <https://www.voltdb.com/>.
- [11] <http://thinkaurelius.github.io/titan/>.
- [12] <http://www.caida.org/data/passive/>.
- [13] <http://www.cray.com/products/analytiscs/cray-graph-engine>.
- [14] Oracle timesten: <http://www.oracle.com/us/products/database/timesten>.
- [15] solidb: <https://teambblue.unicomsi.com/products/solidb>.
- [16] C. R. Aberger, S. Tu, K. Olukotun, and C. Ré. Emptyheaded: A relational engine for graph processing. In *SIGMOD '16*.
- [17] R. Agrawal. Alpha: An extension of relational algebra to express a class of recursive queries. *IEEE Trans. Softw. Eng.*, July 1988.
- [18] R. Chen. Managing massive graphs in relational dbms. In *BIG DATA '13*.
- [19] T. Chondrogianis, P. Bouras, J. Gamper, and U. Leser. Alternative routing: K-shortest paths with limited overlap. In *SIGSPATIAL '15*.

- [20] L. S. Colby. A recursive algebra and query optimization for nested relations. In *SIGMOD '89*.
- [21] S. Dar, R. Agrawal, and H. V. Jagadish. Optimization of generalized transitive closure queries. In *ICDE '91*.
- [22] A. Dave, A. Jindal, L. E. Li, R. Xin, J. Gonzalez, and M. Zaharia. Graphframes: An integrated api for mixing graph and relational queries. In *Proc. of the 4th Int. Workshop on Graph Data Management Experiences and Systems, GRADES '16*.
- [23] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: Sql server’s memory-optimized oltp engine. In *SIGMOD '13*.
- [24] E. W. Dijkstra. A note on two problems in connection with graphs. *Numerical Mathematics*, 1, 1959.
- [25] J. Fan, A. G. S. Raj, and J. M. Patel. The case against specialized graph analytics engines. In *CIDR '15*.
- [26] J. Gao, R. Jin, J. Zhou, J. X. Yu, X. Jiang, and T. Wang. Relational approach for shortest path discovery over large graphs. *Proc. VLDB Endow.*, 5(4), Dec. 2011.
- [27] A. Ghazal, D. Seid, A. Crolotte, and M. Al-Kateb. Adaptive optimizations of recursive queries in teradata. In *SIGMOD '12*.
- [28] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2), June 1993.
- [29] T. J. Green, S. S. Huang, B. T. Loo, and W. Zhou. Datalog and recursive query processing. *Found. Trends databases*, 5(2):105–195, Nov. 2013.
- [30] M. S. Hassan, W. G. Aref, and A. M. Aly. Graph indexing for shortest-path finding over dynamic sub-graphs. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD '16*, pages 1183–1197, 2016.
- [31] A. Jindal, S. Madden, M. Castellanos, and M. Hsu. Graph analytics using vertica relational database. In *BIG DATA '15*.
- [32] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-store: A high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, Aug. 2008.
- [33] D. Kernert, F. Köhler, and W. Lehner. Slacid - sparse linear algebra in a column-oriented in-memory database system. In *SSDBM '14*.
- [34] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD '10*.
- [35] K. Molka and G. Casale. Contention-aware workload placement for in-memory databases in cloud environments. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 2(1), Nov. 2016.
- [36] H. Montaner, F. Silla, H. Fröning, and J. Duato. Memscale: In-cluster-memory databases. In *CIKM '11*.
- [37] M. Paradies, W. Lehner, and C. Bornhövd. Graphite: An extensible graph traversal framework for relational database management systems. In *SSDBM '15*.
- [38] Y. Perez, R. Sosić, A. Banerjee, R. Puttagunta, M. Raison, P. Shah, and J. Leskovec. Ringo: Interactive graph analytics on big-memory machines. In *SIGMOD '15*.
- [39] M. Sadoghi, S. Bhattacharjee, B. Bhattacharjee, and M. Canim. L-Store: A real-time OLTP and OLAP system. In *EDBT '18*.
- [40] S. Sakr, S. Elnikety, and Y. He. Hybrid query execution engine for large attributed graphs. *Inf. Syst.*, 41:45–73, May 2014.
- [41] A. D. Sarma, H. Lee, H. Gonzalez, J. Madhavan, and A. Halevy. Consistent thinning of large geographical data for map visualization. *ACM Trans. Database Syst.*, 38(4), Dec. 2013.
- [42] M. Sarwat, S. Elnikety, Y. He, and G. Kliot. Horton: Online query execution engine for large distributed graphs. In *ICDE '12*.
- [43] M. Sarwat, S. Elnikety, Y. He, and M. F. Mokbel. Horton+: A distributed system for processing declarative reachability queries over partitioned graphs. *Proc. VLDB Endow.*, 6(14), Sept. 2013.
- [44] A. Shahvarani and H.-A. Jacobsen. A hybrid b+tree as solution for in-memory indexing on cpu-gpu heterogeneous computing platforms. In *SIGMOD '16*.
- [45] D. Simmen, K. Schnaitter, J. Davis, Y. He, S. Lohariwala, A. Mysore, V. Sheno, M. Tan, and Y. Xiao. Large-scale graph analytics in aster 6: Bringing context to big data discovery. *Proc. VLDB Endow.*, 7(13), Aug. 2014.
- [46] W. Sun, A. Fokoue, K. Srinivas, A. Kementsietsidis, G. Hu, and G. Xie. Sql-graph: An efficient relational-based property graph store. In *SIGMOD '15*.
- [47] J. R. Thomsen, M. L. Yiu, and C. S. Jensen. Effective caching of shortest paths for location-based services. In *SIGMOD '12*.
- [48] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos. Doulion: Counting triangles in massive graphs with a coin. In *KDD '09*.
- [49] L. Wang, M. Zhou, Z. Zhang, Y. Yang, A. Zhou, and D. Bitton. Elastic pipelining in an in-memory database cluster. In *SIGMOD '16*.
- [50] Z. Wang, S. Mu, Y. Cui, H. Yi, H. Chen, and J. Li. Scaling multicore databases via constrained parallel execution. In *SIGMOD '16*.
- [51] K. Xirogiannopoulos and A. Deshpande. Extracting and analyzing hidden graphs from relational databases. In *SIGMOD '17*.
- [52] K. Xirogiannopoulos, U. Khurana, and A. Deshpande. Graphgen: Exploring interesting graphs in relational data. *Proc. VLDB Endow.*, 8(12), Aug. 2015.
- [53] C. Yao, D. Agrawal, G. Chen, B. C. Ooi, and S. Wu. Adaptive logging: Optimizing logging and recovery costs in distributed in-memory databases. In *SIGMOD '16*.

Sequenced Route Query with Semantic Hierarchy

Yuya Sasaki[†], Yoshiharu Ishikawa[‡], Yasuhiro Fujiwara^{§†}, Makoto Onizuka[†]

[†]Graduate School of Information Science and Technology, Osaka University, Osaka, Japan

[‡]Graduate School of Information Science, Nagoya University, Nagoya, Japan

[§]NTT Software Innovation Center, Tokyo, Japan

sasaki@ist.osaka-u.ac.jp, ishikawa@i.nagoya-u.ac.jp, fujiwara.yasuhiro@lab.ntt.co.jp, onizuka@ist.osaka-u.ac.jp

ABSTRACT

The trip planning query searches for preferred routes starting from a given point through multiple Point-of-Interests (PoI) that match user requirements. Although previous studies have investigated trip planning queries, they lack flexibility for finding routes because all of them output routes that strictly match user requirements. We study trip planning queries that output multiple routes in a flexible manner. We propose a new type of query called *skyline sequenced route (SkySR)* query, which searches for all preferred sequenced routes to users by extending the shortest route search with the semantic similarity of PoIs in the route. Flexibility is achieved by the *semantic hierarchy* of the PoI category. We propose an efficient algorithm for the SkySR query, *bulk SkySR algorithm* that simultaneously searches for sequenced routes and prunes unnecessary routes effectively. Experimental evaluations show that the proposed approach significantly outperforms the existing approaches in terms of response time (up to four orders of magnitude). Moreover, we develop a prototype service that uses the SkySR query, and conduct a user test to evaluate its usefulness.

1 INTRODUCTION

Recently, technological advances in various devices, such as smart phones and automobile navigation systems, have allowed users to obtain real-time location information easily. This has triggered the development of location-based services such as Foursquare, which exploit rich location information to improve service quality. The users of the location-based services often want to find short routes that pass through multiple Points-of-Interest (PoIs); consequently, developing trip planning queries that can find the shortest routes that passes through user-specified categories has attracted considerable attention [4, 10]. If multiple PoI categories, e.g., restaurant and shopping mall, are in an ordered list (i.e., a *category sequence*), the trip planning query searches for a *sequenced route* that passes PoIs that match the user-specified categories in order.

Example 1.1. Figure 1 shows a road network with the following PoIs: “Asian restaurant”, “Italian restaurant”, “Gift shop”, “Hobby shop”, and “Arts&Entertainment (A&E)”. Assume that a user wants to go to an Asian restaurant, an A&E place, and a gift shop in this order from start point v_q . The sequenced route query outputs route R1 because it is the shortest route from v_q that satisfied the user requirements (Asian restaurant, A&E, gift shop).

Existing approaches find the shortest route based on the user query. However, such approaches may find an unexpectedly long

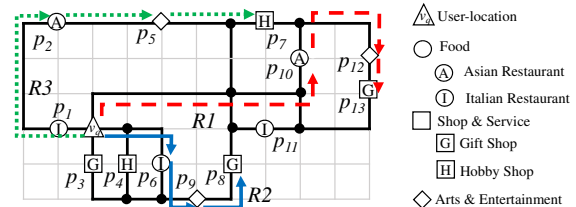


Figure 1: An example of a road network with PoIs

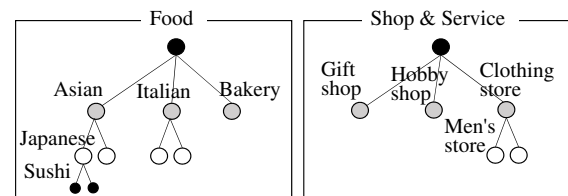


Figure 2: Examples of category trees in Foursquare

route because the found PoIs may be distant from the start point. A major problem with the existing approaches is that they only output routes that *perfectly* match the given categories [5, 14, 16]. To overcome this problem, we introduce flexible similarity matching based on PoI category classification to find shorter routes in a flexible manner. In the real-world, category classification often forms a *semantic hierarchy*, which we refer to as a *category tree*. For example, in Foursquare¹, the “Food” category tree includes “Asian restaurant,” “Italian restaurant,” and “Bakery” as subcategories, and the “Shop &Service” category includes “Gift shop,” “Hobby shop,” and “Clothing store” as subcategories (Figure 2). We employ this semantic hierarchy to evaluate routes in terms of two aspects, i.e., route length and the semantic similarity between the categories of the PoIs in the route and those specified in the user query. As a result, we can find effective sequenced routes that *semantically* match the user requirement based on the semantic hierarchy. For example, in Figure 1, route R2 satisfies the user requirement because it semantically matches the category sequence because Italian and Asian restaurants are in the same category tree. However, this approach may find a significantly large number of sequenced routes because the number of PoIs that flexibly match the given categories increases significantly. To reduce the number of routes to be output, we employ the skyline concept [2], i.e., we restrict ourselves to searching for the routes that are not worse than any other routes in terms of their scores (i.e., numerical values to evaluate the routes). Based on this concept, we propose the *skyline sequenced route (SkySR) query*, which applies the skyline concept to the route length and semantic similarity (i.e., we consider route length and semantic similarity as route scores). Given a start point and a sequence

¹<https://developer.foursquare.com/categorytree>

Table 1: Example routes in New York city

Approach	Distance	Sequenced route
Existing (e.g., [16])	3239 meters	Cupcake Shop → Art Museum → Jazz Club
Proposed	3239 meters	Cupcake Shop → Art Museum → Jazz Club
	1858 meters	Dessert Shop → Art Museum → Jazz Club
	1392 meters	Dessert Shop → Museum → Jazz Club
	823 meters	Dessert Shop → Museum → Music Venue

of PoI categories, a SkySR query searches for sequenced routes that are no worse than any other routes in terms of length and semantic similarity.

Example 1.2. Table 1 shows real-world examples of sequenced routes in New York city where a user plans to go to a cupcake shop, an art museum, and then a jazz club in this order. The existing approaches output a single route that matches the user’s requirement perfectly. The proposed approach can output three additional routes that are shorter than the route found by the existing approach. Note that the additional routes also satisfy the user query semantically. The user can select a preferred route among all the four routes depending on how far he/she does not want to walk or their available time.

The SkySR query can provide effective trip plans; however, it incurs significant computational cost because a large number of routes can match the user requirement. Therefore, the SkySR query requires an efficient algorithm. The challenge is to search for SkySRs efficiently by reducing the search space without sacrificing the exactness of the result. We propose *bulk SkySR* algorithm (BSSR for short) that finds exact SkySRs efficiently. Recall that a feature of SkySRs is that their scores are no worse than those of other sequenced routes. BSSR exploits the branch-and-bound algorithm [9], which effectively prunes unnecessary routes based on the upper and lower bounds of route scores. In addition, to improve efficiency more, we employ four techniques to optimize BSSR. (1) First, we initially find sequenced routes to calculate the upper bound. (2) We tighten the upper bound by arranging the priority queue and (3) tighten the lower bound by introducing minimum distances. (4) we keep intermediate results for later processing, which refer to as *on-the-fly caching*. Our approach significantly outperforms existing approaches in terms of response time (up to four orders of magnitude) without increasing memory usage or sacrificing the exactness of the result.

The main contributions of this paper are as follows.

- We introduce a semantic hierarchy to the route search query, which allows us to search for routes flexibly.
- We propose the *skyline sequenced route (SkySR) query*, which finds all preferred routes related to a specified category sequence with a semantic hierarchy (Section 4).
- We propose an exact and efficient algorithm and its optimization techniques to process SkySR queries (Section 5).
- We discuss variations and extensions of the SkySR query. The SkySR query can be applied to various user requirements and environments (Section 6).
- We demonstrate that the proposed approach works well in terms of response time and memory usage by performing extensive experiments. (Section 7).
- We develop a prototype service that employs the SkySR query and conduct a user test to evaluate usefulness of the SkySR query. (Section 8).

The remainder of this paper is organized as follows. Section 2 introduces related work. Section 3 describes the problem formulation, and Section 4 defines the SkySR query. Section 5 presents the proposed algorithm. In Section 6, we discuss variations and extensions of the SkySR query. Sections 7 and 8 present experiment and user test results, respectively, and Section 9 concludes the paper.

2 RELATED WORK

First, we review trip planning query studies related to the SkySR query. Then, we review some studies related to the skyline operator. To the best of our knowledge, no study has considered a skyline sequenced route; thus, our problem cannot be solved efficiently using existing approaches.

Trip planning: We categorize trip planning queries in Table 2. Note that all existing trip planning queries only output routes that perfectly match the user-specified category sequences. Moreover, since most trip planning queries assume Euclidean distance, they cannot find SkySRs, in which road network distance is assumed. Dai et al. [4] proposed a personalized sequenced route and assumed that PoIs have ratings as well as categories and that users assign weighting factors as preferences. Although this personalized sequenced route considers route lengths and ratings, it only outputs the route that perfectly matches the given categories and has the best score based on lengths, ratings, and preferences. Only the optimal sequenced route (OSR) is applicable to find SkySRs without modification because the OSR and SkySR are based on the same settings (except for scoring). Sharifzadeh et al. [16] proposed two algorithms to find OSRs in road networks: the *Dijkstra-based solution* and the *Progressive Neighbor Exploration (PNE) approach*. The main difference between these algorithms is that the Dijkstra-based solution employs the Dijkstra algorithm to search for PoIs and the PNE approach employs the nearest neighbor search. It has been reported that these algorithms are comparable in terms of performance [16]. Thus, we consider both algorithms to verify the performance of the proposed approach.

Skyline: The skyline operator was proposed previously [2]. Few studies have considered the skyline concept for route searches. Recently, the skyline route (or skyline path) has received considerable attention [1, 6, 8, 13, 17, 18, 20]. A skyline route assumes that edges on road networks are associated with multiple costs, such as distance, travel time, and tolls. Here, the objective is to find skyline routes from a start point to a destination considering these multiple costs. However, since we specify a category sequence rather than a destination, we cannot apply conventional algorithms to find SkySRs. The continuous skyline query in road networks (e.g., [7]) searches for the skyline PoIs for a moving object considering both the PoI category and the distances to the moving object. Because continuous skyline queries search for a single PoI category, these solutions are not applicable to SkySR queries, which obtain routes that pass through multiple PoIs.

3 PRELIMINARIES

Table 3 summarizes the notations used in this paper. We assume a connected graph $G = (\mathbb{V} \cup \mathbb{P}, \mathbb{E})$, where \mathbb{V} , \mathbb{P} , and $\mathbb{E} \subseteq (\mathbb{V} \cup \mathbb{P}) \times (\mathbb{V} \cup \mathbb{P})$ represent the sets of vertices, PoI vertices, and edges, respectively. This graph corresponds to a road network that contains PoIs. The numbers of vertices, PoI vertices, and edges are denoted $|\mathbb{V}|$, $|\mathbb{P}|$, and $|\mathbb{E}|$, respectively. PoI vertex $p \in \mathbb{P}$ is associated with category $c \in \mathbb{C}$, where \mathbb{C} is the set of categories. We denote the category of PoI vertex p as c_p , and assume that

Table 2: Types of trip planning queries.

Type	Distance metrics	Order	Destination	Result	Scores
SkySR (proposed)	Network	Total	Yes or No	Exact	Length and semantic
Optimal sequenced route (OSR) [16]	Euclidean or Network	Total	Yes or No	Exact	Length
Sequenced route [5, 14]	Network	Total	Yes	Exact	Length
Personalized sequenced route [4]	Euclidean	Total	No	Approximate	Length and rating
Trip planning [10]	Euclidean or Network	Non	Yes	Approximate	Length
Multi rule partial sequenced route [3]	Euclidean	Partial	No	Approximate	Length
Multi rule partial sequenced route [11]	Euclidean	Partial	No	Exact	Length
Multi-type nearest neighbor [12]	Euclidean	Non	No	Exact	Length

Table 3: Notations

Symbol	Meaning
\mathbb{V}	Set of vertices
\mathbb{P}	Set of PoI vertices
\mathbb{E}	Set of edges
p	PoI vertex
\mathbb{C}	Set of categories
c	Category
t	Category tree
c_p	Category of PoI vertex p
t_c	Category tree of c
\mathbb{P}_c	Set of PoI vertices associated with c
\mathbb{P}_t	Set of PoI vertices associated with t
\mathbb{S}	Category sequence (sequence of categories)
\mathbf{R}	Route (sequence of PoI vertices)
\mathbb{S}_R	Sequential PoI categories in \mathbf{R}
$l(\mathbf{R})$	Length score of \mathbf{R}
$s(\mathbf{R})$	Semantic score of \mathbf{R}
\mathcal{R}	Set of routes
$\mathcal{E}(\mathbf{R})$	Set of super-routes of \mathbf{R}
\mathcal{S}	Minimal set of sequenced routes
S_q	Category sequence specified by user
v_q	Start point specified by user

each PoI is associated with a single category. Each category is associated with category tree t , and we denote the category tree of category c as t_c . We denote the set of PoI vertices associated with c and the set of PoI vertices associated with the category tree t as \mathbb{P}_c and \mathbb{P}_t , respectively. If a PoI vertex is associated with category c , it is also associated with all ancestor categories of c in t_c . Each edge $e(u_i, u_j)$ in \mathbb{E} is associated with a weight $w(u_i, u_j)$ (≥ 0). The weight can represent either travel duration or distance. Next, we define several terms required to introduce the skyline sequenced route (SkySR).

Definition 3.1. (Category sequence) A category sequence $\mathbf{S} = \langle c_S[1], c_S[2], \dots, c_S[|\mathbf{S}|] \rangle$ is a sequence of categories, where $|\mathbf{S}|$ is the size of \mathbf{S} . $c_S[i] \in \mathbb{C}$ denotes the i -th category in \mathbf{S} . A *super-category sequence* of \mathbf{S} is a category sequence where each i -th category is either $c_S[i]$ or an ancestor of $c_S[i]$ ($1 \leq i \leq |\mathbf{S}|$) in the category tree.

Definition 3.2. (Route) A route $\mathbf{R} = \langle p_R[1], \dots, p_R[|\mathbf{R}|] \rangle$ is a sequence of PoI vertices in a road network, where $p_R[i] \in \mathbb{P}$ and $|\mathbf{R}|$ denote the i -th PoI vertex in \mathbf{R} and the size of \mathbf{R} , respectively. \mathbb{S}_R denotes the category sequence of \mathbf{R} (i.e., $\langle c_{p_R[1]}, \dots, c_{p_R[|\mathbf{R}|]} \rangle$). In addition, we define a *super-route* of \mathbf{R} as an extended route of \mathbf{R} , such as $\langle \mathbf{R}, p_i, p_j, \dots \rangle$. In other words, a super-route of \mathbf{R} is obtained by adding a sequence of PoI vertices to the end of \mathbf{R} . \mathcal{R} and $\mathcal{E}(\mathbf{R})$ denote a set of routes and a set of super-routes of \mathbf{R} , respectively. Moreover, given a route $\mathbf{R} = \langle p_R[1], \dots, p_R[|\mathbf{R}|] \rangle$ and a PoI vertex p , we define $\mathbf{R} \oplus p = \langle p_R[1], \dots, p_R[|\mathbf{R}|], p \rangle$.

Definition 3.3. (Category similarity) Given two categories c and c' , the similarity $sim(c, c') \in [0, 1]$ is calculated by an arbitrary function such as the *Wu and Palmer similarity* or path

length [15, 19]. We assume the following relations in the similarity.

- c is irrelevant to c' if both exist in different category trees; thus, we obtain $sim(c, c') = 0$.
- c *semantically matches* c' if c and c' are in the same category tree; thus, we obtain $0 < sim(c, c') \leq 1$.
- c *perfectly matches* c' if c and c' are the same; thus, we obtain $sim(c, c') = 1$.

Note that a semantic match subsumes a perfect match.

We define a *sequenced route* using the above definitions. The difference between our definition of *sequenced route* and the previous definition [16] is that we consider category similarity.

Definition 3.4. (Sequenced route) Given category sequence $\mathbf{S} = \langle c_S[1], \dots, c_S[|\mathbf{S}|] \rangle$, $\mathbf{R} = \langle p_R[1], \dots, p_R[|\mathbf{R}|] \rangle$ is a *sequenced route* of category sequence \mathbf{S} if and only if it satisfies (i) $|\mathbf{R}| = |\mathbf{S}|$, (ii) $c_S[i]$ semantically matches $c_{p_R[i]}$ for all i such that $1 \leq i \leq |\mathbf{S}|$, and (iii) all PoI vertices in \mathbf{R} differ each other.

Definition 3.5. (Route scores) Given category sequence \mathbf{S} and vertex v as a start point, we define two scores for route \mathbf{R} : *length score* $l(\mathbf{R}) \in [0, \text{inf}]$ and *semantic score* $s(\mathbf{R}) \in [0, 1]$. We define the length score $l(\mathbf{R})$ as follows:

$$l(\mathbf{R}) = D(v, p_R[1]) + \sum_{i=1}^{|\mathbf{R}|-1} D(p_R[i], p_R[i+1]), \quad (1)$$

where $D(u_i, u_j)$ denotes the smallest weight sum of the edges on the routes between vertices (or PoIs) u_i and u_j . The semantic score $s(\mathbf{R})$ is calculated by an aggregation function f as follows:

$$s(\mathbf{R}) = f(h_1, h_2, \dots, h_{|\mathbf{R}|}), \quad (2)$$

where h_i denotes $sim(c_S[i], c_{p_R[i]})$. We assume that, if all $h_i = 1$, $s(\mathbf{R}) = 0$, i.e., if all PoI vertices in a route perfectly match the categories, the semantic score of the given route is 0. We also assume that $\underline{s}(\mathbf{R})$ is the possible minimum semantic score of \mathbf{R} when it is a sequenced route. Without loss of generality, preferred routes have small length and semantic score.

4 THE SKYLINE SEQUENCED ROUTE QUERY

Here, we define the SkySR query. Intuitively, a SkySR is a potential route that may be the best route related to the user's requirement. A potential route is a route that is not *dominated* by any other routes; the notion of *dominance* is used in the *skyline operator* [2]. We define dominance for sequenced routes and SkySR query in the following.

Definition 4.1. (Dominance) Let \mathcal{R} be the set of all sequenced routes starting from point v for category sequence \mathbf{S} . For two sequenced routes $\mathbf{R}, \mathbf{R}' \in \mathcal{R}$, we say that \mathbf{R} dominates \mathbf{R}' if we have (i) $l(\mathbf{R}) < l(\mathbf{R}')$ and $s(\mathbf{R}) \leq s(\mathbf{R}')$ or (ii) $s(\mathbf{R}) < s(\mathbf{R}')$ and $l(\mathbf{R}) \leq l(\mathbf{R}')$. If two sequenced routes have the same length and

semantic scores, the routes are *equivalent* in the dominance, and a set of sequenced routes is *minimal* if it has no equivalent routes.

Definition 4.2. (SkySR query) Given vertex v_q as a start point and category sequence S_q , a skyline sequenced route is a sequenced route not dominated by other routes. Let \mathcal{R} be the set of all sequenced routes from start point v_q for category sequence S_q , and let \mathcal{S} be a *minimal* set of the sequenced routes. The SkySR query returns \mathcal{S} that includes sequenced routes such that all $\mathbf{R} \in \mathcal{S}$ are SkySRs and all $\mathbf{R}' \in \mathcal{R} \setminus \mathcal{S}$ are dominated by or equivalent to some of $\mathbf{R} \in \mathcal{S}$.

An naive solution to find SkySRs is to first enumerate SkySR candidates by iteratively executing OSR queries for any super-category sequences of S_q and then check the dominance among the routes. The number of super-category sequences of S_q increases exponentially as the depth of the category in the category tree and the size of S_q increase. Thus, although OSR algorithms can find a sequenced route efficiently, we must repeat many searches. As a result, the naive solution needs significantly high computational cost to find SkySRs.

5 PROPOSED ALGORITHM

In this section, we present the proposed approach, which we refer to as the *bulk SkySR algorithm* (BSSR), that finds SkySRs efficiently. Section 5.1 presents the BSSR design policy, and Section 5.2 explains the BSSR procedure. In Section 5.3, we propose optimization techniques for BSSR. We also theoretically analyze its performance in Section 5.4. Finally, we show a running example of BSSR in Section 5.5. In Section 5, we assume undirected graphs in which each PoI vertex is associated with only one category and that users give sequences of single PoI categories. However, in a real application, the graphs would be directed graphs, each PoI vertex would be associated with multiple categories, and users may specify complex categories. Section 6 describes how we handle the above conditions.

5.1 Design Policy

Our idea to improve efficiency is to find sequenced routes simultaneously (i.e., by searching sequenced routes in bulk) in order to reduce the search space. We have two choice as the basis for our approach; Dijkstra-based or nearest neighbor-based approaches [16]. We use the Dijkstra-based approach as the basis of our algorithm. Recall that a SkySR query has two scores for a route, i.e., length and semantic scores. To find all SkySRs, we must find routes that have small category scores even if the routes have large length scores. However, PoIs that are included in the routes with small category scores could be distant from the start point. Although the nearest neighbor-based approach finds the closest PoIs, it cannot efficiently find such PoIs. On the other hand, the Dijkstra-based approach searches for all PoI vertices that match a PoI category. Therefore, the Dijkstra-based approach is more suitable for the SkySR query than the nearest neighbor-based approach.

Although our approach finds sequenced routes simultaneously, it entails a large number of executions of the Dijkstra algorithm. This is because, since the number of PoI candidates increases, a large number of possible routes increases. The search space does not become small effectively. To effectively reduce the search space, we exploit the branch-and-bound algorithm, which uses the upper and lower bounds of a branch of the search space to solve an optimization problem effectively. With BSSR, each branch corresponds to each route. For the upper and lower

bounds, we compute the bounds during finding the set of SkySRs. Specifically, we compute the upper bound of a route from the already found sequenced routes, and we compute the lower bound from the current searched route (i.e., not a sequenced route yet). With the upper and lower bounds, we can safely prune unnecessary routes to improve efficiency.

To further increase efficiency, we propose optimization techniques for BSSR. In order to exploit the branch-and-bound algorithm, it is necessary to initialize the upper bound. Thus, we first search for a sequenced route to initialize the upper bound. However, it may take high computational cost to find a sequenced route. Therefore, we propose a *nearest neighbor-based initial search method* (NNinit) that finds sequenced routes efficiently by greedily finding PoI vertices. In addition, to effectively update the upper bound, we assign a priority to each route and use the priority queue to efficiently find routes that are likely to give an effective upper bound. To compute the lower bound, we compute the possible minimum distance and add it to the length score of a route to safely prune unnecessary routes. Moreover, to avoid executing the Dijkstra algorithm iteratively from the same vertices, we materialize search results of the Dijkstra algorithm and reuse them to search the PoI vertices. By using BSSR with optimization techniques, we can perform the SkySR query efficiently.

5.2 Bulk SkySR algorithm

Bulk SkySR algorithm (BSSR) finds all SkySRs by finding simultaneously sequenced routes with checking dominance on demand. The naive solution must execute OSR queries for all super-category sequences of S_q one by one because it only searches for the PoIs that perfectly match the given category. In contrast, BSSR searches for all PoIs that semantically match the given category.

The basic process of BSSR is simple as shown in Algorithm 1: (i) start searching the PoI vertices that match the first category from start point v_q and insert the route found into priority queue Q_b which stores all found routes (line 4), (ii) fetch a route from Q_b (line 6), (iii) search for the next PoI vertices that semantically match the next category c_d from PoI vertex p_d which is the end of the fetched route, and insert the fetched route with each of the found PoI vertices into Q_b (lines 7–9), and (iv) if Q_b is not empty, return to (ii), otherwise output the minimal set of sequenced route \mathcal{S} (line 10). In steps (i) and (iii), we find PoI vertices from the end of the fetched route using a Dijkstra algorithm modified for the SkySR query as described in Section 5.2.2.

Algorithm 1: Bulk SkySR algorithm

```

1 procedure BSSR( $v_q, S_q$ )
2  $\mathcal{S} \leftarrow \phi$ ;
3 priority_queue  $Q_b \leftarrow \phi$ ;
4  $\text{mDijkstra}(\phi, c_S[1], v_q, Q_b, \mathcal{S})$ ;
5 while  $Q_b$  is not empty do
6    $\mathbf{R} \leftarrow Q_b.\text{dequeue}()$ ;
7    $c_d \leftarrow c_S[|\mathbf{R}| + 1]$ ;
8    $p_d \leftarrow p_{\mathbf{R}}[|\mathbf{R}|]$ ;
9    $\text{mDijkstra}(\mathbf{R}, c_d, p_d, Q_b, \mathcal{S})$ ;
10 return  $\mathcal{S}$ ;
11 end procedure

```

5.2.1 Branch-and-bound. We search for sequenced routes simultaneously to reduce the search space. Our idea to safely reduce the search space is to exploit the branch-and-bound algorithm, which can reduce unnecessary search space. This section

describes the theoretical background of using the branch-and-bound algorithm. We use the following three lemmas to reduce the search space:

LEMMA 5.1. *Let \mathcal{S} be a minimum set of sequenced routes while searching for SkySRs and \mathcal{S}' be the minimum set of sequenced routes after finding SkySRs. If sequenced route \mathbf{R} is dominated by a sequenced route in \mathcal{S} , \mathbf{R} cannot be included in \mathcal{S}' .*

proof: From Definition 4.2, we search for a set of SkySRs, which are not dominated by the other sequenced routes. If we find a sequenced route not dominated by any sequenced routes in \mathcal{S} , we update \mathcal{S} by inserting the new sequenced route and deleting a sequenced route dominated by the new one. Therefore, any sequenced routes in \mathcal{S} after the update are not dominated by any sequenced routes in \mathcal{S} prior to the update. As a result, sequenced routes in \mathcal{S}' are not dominated by any sequenced routes in \mathcal{S} . In other words, \mathbf{R} is not included in \mathcal{S}' if we have sequenced route \mathbf{R}' in \mathcal{S} such that $l(\mathbf{R}') \leq l(\mathbf{R})$ and $s(\mathbf{R}') \leq s(\mathbf{R})$. \square

LEMMA 5.2. *Let $\mathcal{E}(\mathbf{R})$ be a set of super-routes of \mathbf{R} starting from the same start point. For any route \mathbf{R}' in $\mathcal{E}(\mathbf{R})$, the length and semantic scores $l(\mathbf{R}')$ and $s(\mathbf{R}')$ cannot be less than $l(\mathbf{R})$ and $\underline{s}(\mathbf{R})$, respectively.*

proof: Let \mathbf{R}' be a route included in $\mathcal{E}(\mathbf{R})$. Since we have $D(u_i, u_j) \geq 0$, the following property holds for a route \mathbf{R} from Equation (1) of Definition 3.5.

$$\begin{aligned} & D(v_q, p_{R'}[1]) + \sum_{i=1}^{|\mathbf{R}'|-1} D(p_{R'}[i], p_{R'}[i+1]) \\ &= D(v_q, p_R[1]) + \sum_{i=1}^{|\mathbf{R}'|-1} D(p_R[i], p_R[i+1]) \\ & \quad + \sum_{i=|\mathbf{R}|}^{|\mathbf{R}'|-1} D(p_{R'}[i], p_{R'}[i+1]) \\ & \geq D(v_q, p_R[1]) + \sum_{i=1}^{|\mathbf{R}'|-1} D(p_R[i], p_R[i+1]). \end{aligned}$$

Therefore, we have $l(\mathbf{R}) \leq l(\mathbf{R}')$. $\underline{s}(\mathbf{R})$ is the possible minimum semantic score of \mathbf{R} when it becomes a sequenced route. Thus, even if PoI vertices are added to \mathbf{R} , we have $\underline{s}(\mathbf{R}) \leq s(\mathbf{R}')$. As a result, we have $l(\mathbf{R}) \leq l(\mathbf{R}')$ and $\underline{s}(\mathbf{R}) \leq s(\mathbf{R}')$. \square

In terms of the branch-and-bound algorithm, Lemma 5.1 and 5.2 give us the upper and lower bounds of the scores of a route, respectively. We can prune routes according to the following lemma.

LEMMA 5.3. (pruning condition) *If (i) \mathbf{R} is a sequenced route included in the set \mathcal{S} of sequenced routes and (ii) $l(\mathbf{R}) \leq l(\mathbf{R}')$ and $s(\mathbf{R}) \leq \underline{s}(\mathbf{R}')$, any routes in $\mathcal{E}(\mathbf{R}')$ cannot be included in \mathcal{S} .*

proof: If we have $l(\mathbf{R}) \leq l(\mathbf{R}')$ and $s(\mathbf{R}) \leq \underline{s}(\mathbf{R}')$, \mathbf{R}' is not included in \mathcal{S} (Lemma 5.1). From Lemma 5.2, the scores of \mathbf{R}' cannot become less than $l(\mathbf{R}')$ and $\underline{s}(\mathbf{R}')$ even if we expand \mathbf{R}' . Therefore, any routes in $\mathcal{E}(\mathbf{R}')$ cannot be included in \mathcal{S} because \mathbf{R}' is dominated by or equivalent to the sequenced route with $l(\mathbf{R})$ and $s(\mathbf{R})$. \square

Lemma 5.3 gives us the length score *threshold* for a route, and, if the length score of a route is greater than this threshold, we can prune the given route. We define the length score threshold of a route as follows:

Definition 5.4. The threshold $\bar{l}(\mathbf{R})$ of the length score of route \mathbf{R} is given by the following equation:

$$\bar{l}(\mathbf{R}) = \min_{\mathbf{R}' \in \mathcal{S}} \{l(\mathbf{R}') | \underline{s}(\mathbf{R}) \geq s(\mathbf{R}')\}. \quad (3)$$

If $\bar{l}(\mathbf{R}) \leq l(\mathbf{R})$, we can safely prune \mathbf{R} because it cannot be included in the result. Thus, we can reduce the search space

without sacrificing the exactness of the result. Equation (3) has a small computation cost because \mathcal{S} includes only a small number of sequenced routes as shown in Section 7.

5.2.2 The modified Dijkstra Algorithm. We search the next PoI vertices that semantically match the next PoI category using the modified Dijkstra algorithm. The modified Dijkstra algorithm can prune unnecessary routes based on Lemma 5.3. Moreover, based on the following lemma, it terminates unnecessary traversal of the graph and avoids inserting unnecessary routes.

LEMMA 5.5. *Let $\mathbf{R} = \langle p_R[1], \dots, p_R[i], p_R[i+1], p_R[i+2], \dots, p_R[|\mathbf{R}|] \rangle$ be a route and $p_{i:i+1}$ be a PoI vertex on a path between $p_R[i]$ and $p_R[i+1]$. Route \mathbf{R} must be dominated by or equivalent to another route if we have $\text{sim}(c_S[i+1], c_{p_{i:i+1}}) \geq \text{sim}(c_S[i+1], c_{p_R[i+1]})$.*

proof: Let $\mathbf{R}' = \langle p_{R'}[1], \dots, p_{R'}[i], p_{i:i+1}, p_{R'}[i+2], \dots, p_{R'}[|\mathbf{R}'|] \rangle$ be a route such that the difference between \mathbf{R} and \mathbf{R}' is only in $p_{i:i+1}$ and $p_R[i+1]$. Since the PoI vertex $p_{i:i+1}$ is on the path between $p_R[i]$ and $p_R[i+1]$, we have $l(\mathbf{R}) \geq l(\mathbf{R}')$ based on triangle inequality (i.e., $D(p_{i:i+1}, p_R[i+1]) + D(p_R[i+1], p_R[i+2]) \geq D(p_{i:i+1}, p_R[i+2])$). Moreover, if $\text{sim}(c_S[i+1], c_{p_{i:i+1}}) \geq \text{sim}(c_S[i+1], c_{p_R[i+1]})$, we have $s(\mathbf{R}) \geq s(\mathbf{R}')$. Therefore, \mathbf{R} is dominated by or equivalent to \mathbf{R}' because $l(\mathbf{R}) \geq l(\mathbf{R}')$ and $s(\mathbf{R}) \geq s(\mathbf{R}')$. \square

Lemma 5.5 gives us two properties for the SkySR query: (i) even if we find a PoI vertex that passes through another PoI vertex that has a better category similarity, we can ignore the PoI vertex, and (ii) if we find a PoI vertex that perfectly matches the given category, we do not need to traverse the graph through the PoI vertex. As a result, using Lemma 5.3 and 5.5, we can efficiently find the next PoI vertices.

Algorithm 2 shows the pseudocode for the modified Dijkstra algorithm, which is used to find PoI vertices that semantically match c_d from p_d . In priority queue Q_d for the modified Dijkstra algorithm, the top vertex is the closest vertex to p_d . The queue is initialized to p_d (line 3). The closest vertex to p_d is dequeued from Q_d (line 5). \mathbf{R}_t is a route expanded from \mathbf{R}_d , which is \mathbf{R}_d with fetched vertex u (line 7). If the length score of \mathbf{R}_t is greater than or equal to the threshold of \mathbf{R}_d , the modified Dijkstra algorithm terminates the process (Lemma 5.3) (line 8). We check whether (i) u semantically matches c_d and (ii) u does not proceed through another PoI vertex whose category similarity is greater than or equal to that of u (line 9). If we satisfy the above conditions and the length score of \mathbf{R}_t is less than its threshold (line 10), we insert \mathbf{R}_t into the priority queue or the set of sequenced routes (lines 10–12). Otherwise, we skip the process to insert \mathbf{R}_t (Lemma 5.3 and 5.5). The neighbor vertices of u are inserted into Q_d unless u perfectly matches c_d (Lemma 5.5) (lines 13–17).

5.3 Optimization techniques

In this section, we propose four optimization techniques for BSSR. Section 5.3.1 explains an initial search for sequenced routes and proposes NNinit. We then explain tightening the upper and the lower bounds in Section 5.3.2 and Section 5.3.3, respectively. Furthermore, in Section 5.3.4 we propose an *on-the-fly caching technique* to reuse previous search results of the modified Dijkstra algorithm.

5.3.1 Initial search. We prune unnecessary routes efficiently using the branch-and-bound algorithm. However, we cannot calculate the threshold of \mathbf{R} if there are no sequenced routes in \mathcal{S} whose semantic scores are not greater than that of $\underline{s}(\mathbf{R})$ based

Algorithm 2: Modified Dijkstra algorithm to find the next PoI vertices matching c_d from p_d

```

1 procedure mDijkstra( $R_d, c_d, p_d, Q_b, S$ )
2  $dist[u] = \inf$  for all  $u \in \mathbb{V} \cup \mathbb{P}, dist[p_d] = 0$ ;
3 priority_queue  $Q_d \leftarrow \{p_d\}$ ;
4 while  $Q_d$  is not empty do
5    $u \leftarrow Q_d.dequeue$ ;
6   if  $u$  is already visited then continue;
7    $R_t \leftarrow R_d \oplus u$ ;
8   if  $l(R_t) \geq \bar{l}(R_d)$  then break;
9   if  $u \in \mathbb{P}_{t_{c_d}}$  and  $u$  is not through the PoI vertex whose category
similarity is higher than that of  $u$  then
10    if  $l(R_t) < \bar{l}(R_t)$  then
11      if  $R_t$  is a sequenced route then  $S.update(R_t)$ ;
12      else  $Q_b.enqueue(R_t)$ ;
13    if  $u \notin \mathbb{P}_{c_d}$  then
14      for each  $u'$  for  $e(u, u') \in \mathbb{E}$  do
15        if  $dist[u] + w(u, u') < dist[u']$  then
16           $dist[u'] = dist[u] + w(u, u')$ ;
17           $Q_d.enqueue(u')$ ;
18 end procedure

```

on Equation (3). Therefore, initially, we search for the sequenced route whose semantic score is 0. However, the length score of the sequenced route can be large if its semantic score is 0. To tighten the threshold, we also search for sequenced routes whose semantic scores are greater than 0 because the length scores of them are less than that of the sequenced route with a semantic score of 0. We initially find several sequenced routes to tighten the upper bound.

We propose NNinit, which searches for several sequenced routes efficiently. NNinit performs a nearest neighbor search repeatedly to find PoI vertices that perfectly match the given categories. With this process, we can find a sequenced route whose semantic score is 0. Moreover, NNinit can find the PoI vertex that semantically matches the given category during the nearest neighbor search. When we find the last visited PoI vertex, we may find PoI vertices that semantically match the last category in S_q . Therefore, we can obtain sequenced routes whose semantic scores are greater than 0 and length scores are small. As a result, NNinit can find several sequenced routes without incurring additional cost, and one of the sequenced routes has a semantic score of 0.

We present the pseudocode for NNinit in Algorithm 3. Here, priority queue Q is initialized to start point v_q (line 3). NNinit repeats the Dijkstra algorithm $|S_q|$ times to find sequenced routes (line 4). The Dijkstra algorithm is executed to search for the closest PoI vertex that perfectly matches $c_{S_q}[i]$ from the initial vertex (the first initial vertex is v_q) (lines 5–19). Here, the closest vertex to the initial vertex is dequeued from Q (line 7). If the algorithm finds a PoI vertex that perfectly matches $c_{S_q}[i]$, this vertex is added to R and Q is initialized to the PoI vertex (lines 12–15). When it finds the last PoI vertex that semantically matches $c_{S_q}[|S_q|]$, it inserts the sequenced route into S (lines 9–11). Finally, we obtain a set of sequenced routes, and one of the sequenced routes in S has a semantic score of 0.

Example 5.6. We show an example of NNinit using Example 1.1, which searches an Asian restaurant, an A&E place, and a gift shop in this order from start point v_q . NNinit executes the Dijkstra algorithm three times because the size of category sequence is three. First, NNinit searches PoI vertices that perfectly match Asian restaurant from v_q . Then, it finds p_2 that is the closest PoI

Algorithm 3: Initial search for finding sequenced routes with a small cost

```

1 procedure NNinit( $v_q, S_q$ )
2  $S \leftarrow \phi, R \leftarrow \phi$ ;
3 priority_queue  $Q \leftarrow \{v_q\}$ ;
  /* execute Dijkstra algorithm  $|S_q|$  times */
4 for  $i : 1$  to  $|S_q|$  do
5    $dist[u] = \inf$  for all  $u \in \mathbb{V} \cup \mathbb{P}, dist[Q.top] = 0$ ;
6   while  $Q$  is not empty do
7      $u \leftarrow Q.dequeue$ ;
8     if  $u$  is already visited then continue;
9     if  $i = |S_q|$  and  $u \in \mathbb{P}_{t_{c_{S_q}[i]}}$  then
10       $R' \leftarrow R \oplus u$ ;
11       $S.update(R')$ ;
12     if  $u \in \mathbb{P}_{c_{S_q}[i]}$  then
13       $R \leftarrow R \oplus u$ ;
14       $Q \leftarrow \{u\}$ ;
15      break;
16     for each  $u'$  for  $e(u, u') \in E$  do
17       if  $dist[u] + w(u, u') < dist[u']$  then
18          $dist[u'] = dist[u] + w(u, u')$ ;
19          $Q.enqueue(u')$ ;
20 return  $S$ ;
21 end procedure

```

that perfectly match Asian restaurant to v_q . Next, it searches the closest PoI vertex that perfectly matches A&E to p_2 and then finds p_5 . From the next search, NNinit inserts sequenced routes to S when it finds PoI vertices that semantically match gift shop. NNinit finds p_7 whose category is Shop&Service (i.e., semantically match) and thus inserts $\langle p_2, p_5, p_7 \rangle$ to S . After finding p_7 , it finds p_8 that perfectly matches gift shop and inserts $\langle p_2, p_5, p_8 \rangle$ to S . Finally NNinit returns S including $\{\langle p_2, p_5, p_8 \rangle, \langle p_2, p_5, p_7 \rangle\}$. The length score of $\langle p_2, p_5, p_7 \rangle$ is 12, which is less than the length score of $\langle p_2, p_5, p_8 \rangle$ of 15.

5.3.2 Tightening upper bound: Arranging routes in the priority queue. We use the upper bound to prune unnecessary routes. The upper bound is computed from the obtained sequenced routes. To tighten the upper bound, it is important to efficiently find sequenced routes that have small length and semantic scores. BSSR extends a route at the top of the priority queue to search for a sequenced route, as shown in Algorithm 1. Note that priority queues in existing algorithms conventionally consider only distances (i.e., a distance-based priority queue). If we use a distance-based priority queue, BSSR preferentially extends a route with a small length score. Although we must increase the size of a route to $|S_q|$ to find a sequenced route, a route that has a small length score likely has a small size. Therefore, it is difficult to search for sequenced routes efficiently using a distance-based priority queue.

To search for sequenced routes efficiently, we preferentially extend a route that has a large size. Here, since many routes in the priority queue could have the same size, we must consider an additional priority, which is expected to affect performance. If multiple routes in the priority queue are the same size, we preferentially extend the route with the smallest semantic score. We can reduce the search space by searching for sequenced routes in ascending order of semantic score. Moreover, if routes are the same size and have the same semantic score, we preferentially extend the route with the smallest length score. As a result, we can efficiently obtain sequenced routes with small length and semantic scores.

5.3.3 Tightening lower bound: Possible minimum length score.

As described in Section 5.2.1, we use the length scores of routes as the lower bound, i.e., we prune a route if the length score of the route is not less than the threshold. Note that the length score of the route increases as the route size increases. This indicates that it is difficult to prune routes before the route size increases. Our approach to tighten the lower bound of the route is to estimate the increase of the length score. However, if we carelessly estimate a future length score, we may sacrifice the exactness of the result.

The basic idea of this estimation is to calculate the *possible minimum distance*. Here, we compute the smallest distance among any pair of PoI vertices in sets of PoI vertices. We use the following two minimum distances, *semantic-match minimum distance* \underline{l}_s and *perfect-match minimum distance* \underline{l}_p :

Definition 5.7. (minimum distance) The semantic-match minimum distance \underline{l}_s and perfect-match minimum distance \underline{l}_p are given by the following equations:

$$\underline{l}_s(\mathbf{R}) = \sum_{i=|\mathbf{R}|-1}^{|\mathbf{S}_q|-1} \underline{l}_s[i], \text{ where } \underline{l}_s[i] = \min_{p_i \in \mathbb{P}_{t_i}, p_{i+1} \in \mathbb{P}_{t_{i+1}}} D(p_i, p_{i+1}). \quad (4)$$

$$\underline{l}_p(\mathbf{R}) = \sum_{i=|\mathbf{R}|-1}^{|\mathbf{S}_q|-1} \underline{l}_p[i], \text{ where } \underline{l}_p[i] = \min_{p_i \in \mathbb{P}_{t_i}, p_{i+1} \in \mathbb{P}_{c_{i+1}}} D(p_i, p_{i+1}). \quad (5)$$

In Equations (4) and (5), \mathbb{P}_{t_i} and \mathbb{P}_{c_i} denote the set of PoI vertices associated with a category tree of $c_{S_q}[i]$ and the set of PoI vertices whose category is $c_{S_q}[i]$, respectively.

We compute the semantic-match minimum distance based on the distance to the PoI vertices that semantically match the next category. We can safely add the semantic-match minimum distance to the current length score without restriction. However, the semantic-match minimum distance is much less than the threshold. Thus, it could be difficult to improve pruning performance; thus, we use the perfect-match minimum distance to increase pruning performance. The perfect-match minimum distance is computed based on the distance to the PoI vertices that perfectly match the next category. We can improve pruning performance using the perfect-match minimum distance compared to the semantic-match minimum distance because the perfect-match minimum distance is much greater than the semantic-match minimum distance; therefore, the perfect-match minimum distance tightens the lower bound more than the semantic-match minimum distance. However, we can use the perfect-match minimum distance only in a special case, i.e., where a route must pass only PoIs that perfectly match the given categories so as not to be dominated. The perfect-match minimum distance works well if the number of sequenced route in \mathbf{S} is large because the constraint is usually satisfied by increasing the number of sequenced route in \mathbf{S} .

LEMMA 5.8. *Let \mathbf{R}' and \mathbf{R}'' be sequenced routes in \mathbf{S} and \mathbf{R} be a route such that (i) $l(\mathbf{R}) \geq l(\mathbf{R}')$ and $s(\mathbf{R}) < s(\mathbf{R}')$ and (ii) $l(\mathbf{R}) < l(\mathbf{R}'')$ and $s(\mathbf{R}) \geq s(\mathbf{R}'')$. Let δ be the minimum increment of a semantic score². We can prune \mathbf{R} if we have (a) $l(\mathbf{R}) \geq l(\mathbf{R}')$ and $s(\mathbf{R}) + \delta \geq s(\mathbf{R}')$ and (b) $l(\mathbf{R}) + \underline{l}_p(\mathbf{R}) \geq l(\mathbf{R}'')$ and $s(\mathbf{R}) \geq s(\mathbf{R}'')$.*

proof: First, we consider case (a). If we have $l(\mathbf{R}) \geq l(\mathbf{R}')$ and $s(\mathbf{R}) + \delta \geq s(\mathbf{R}')$, \mathbf{R} is dominated by or equivalent to \mathbf{R}' if its semantic score increases. Therefore, \mathbf{R} must only pass through PoI vertices that perfectly match the given categories not to be dominated. If \mathbf{R} passes through only PoI vertices that perfectly

²The least increase of the semantic score is computed from the category tree. Specifically, we can compute the least increase from the category that is most similar (but not equal) to the next category.

match the given categories, the length score of \mathbf{R} increases by at least $\underline{l}_p(\mathbf{R})$. For case (b), if we have $l(\mathbf{R}) + \underline{l}_p(\mathbf{R}) \geq l(\mathbf{R}'')$ and $s(\mathbf{R}) \geq s(\mathbf{R}'')$, \mathbf{R} is dominated by or equivalent to \mathbf{R}'' if its length score increases by $\underline{l}_p(\mathbf{R})$. As a result, if we have two routes \mathbf{R}' and \mathbf{R}'' , such as (i) $l(\mathbf{R}) \geq l(\mathbf{R}')$ and $s(\mathbf{R}) + \delta \geq s(\mathbf{R}')$ and (ii) $l(\mathbf{R}) + \underline{l}_p(\mathbf{R}) \geq l(\mathbf{R}'')$ and $s(\mathbf{R}) \geq s(\mathbf{R}'')$, \mathbf{R} is dominated by or equivalent to at least one of \mathbf{R}' and \mathbf{R}'' . \square

To compute the estimation of the lower bound, we compute two types of possible minimum distances \underline{l}_s and \underline{l}_p . A naive approach computes all minimum distances from the PoI vertices that semantically match $c_{S_q}[i]$ to $c_{S_q}[i+1]$ for $1 \leq i \leq |\mathbf{S}_q| - 1$ by iteratively executing the Dijkstra algorithm. However, this has a high computational cost. To reduce the cost, we execute a *multi-source multi-destination Dijkstra algorithm*. In this algorithm, all start points are inserted into the same priority queue. Then, the algorithm dequeues vertices in the same manner as the conventional Dijkstra algorithm. Here, the process is terminated if the top of the priority queue becomes one of the destinations. This approach only needs $|\mathbf{S}_q| - 1$ times to compute the possible minimum distance. The multi-source multi-destination Dijkstra algorithm guarantees the minimum distance by the following lemma:

LEMMA 5.9. *The multi-source multi-destination Dijkstra algorithm guarantees the minimum distance from the start points to the destinations.*

proof: We first insert multiple start points into the priority queue, and their distances from the start points are initialized as 0. If we find a vertex, it is inserted into the queue and the distance to the vertex is updated from the closest start point to the vertex. The vertex with the smallest distance from the start point in the priority queue is dequeued from the priority queue. If the top vertex in the priority queue is one of the destinations, there are no destinations with smaller distance than the top one. Therefore, we can guarantee the minimum distance from the start points to the destinations. \square

Algorithm 4 shows the pseudocode to compute the semantic-match minimum distance. The estimation of the lower bound is executed after line 4 in Algorithm 1. Here, we initialize \mathbb{P}_1 and \mathbb{P}_{i+1} (lines 3–4). $\bar{l}(\phi)$ denotes the threshold for a route whose semantic score is 0. The difference between computing the semantic-match and perfect-match minimum distances is whether the PoI vertices in \mathbb{P}_{i+1} semantically or perfectly match the given category.

Example 5.10. We show an example to compute the semantic-match minimum distance using Example 1.1. \mathbb{P}_1 , \mathbb{P}_2 , and \mathbb{P}_3 include $\{p_1, p_2, p_6, p_{10}, p_{11}\}$, $\{p_5, p_9, p_{12}\}$, and $\{p_3, p_4, p_7, p_8, p_{13}\}$, respectively. First, PoI vertices in \mathbb{P}_1 are inserted to priority queue Q , and the set of destinations is \mathbb{P}_2 . By processing the Dijkstra algorithm, we compute possible minimum distance $\underline{l}_s[1] = 2$ (from p_6 to p_9). Next, we search PoI vertices that semantically match A&E to gift shop. Then, we compute $\underline{l}_s[2] = 1$ (from p_{12} to p_{13}). Finally, we obtain semantic-match minimum distance $\underline{l}_s = \{2, 1\}$. We can compute the perfect-match minimum distance in the same way and obtain $\underline{l}_p = \{3, 1\}$, which is greater than \underline{l}_s .

5.3.4 Reuse of the temporal result: On-the-fly caching technique. Although BSSR efficiently prunes unnecessary routes, it may iteratively execute the modified Dijkstra algorithm at the same vertex because, in Algorithm 1 (line 8), p_d could be the

Algorithm 4: Computing possible minimum distance

```
1 procedure EstimationLowerbound( $v_q, S_q$ )
2 for  $i : 1$  to  $|S_q| - 1$  do
3    $\mathbb{P}_i \leftarrow \{p | p \in \mathbb{P}_{c_{S_q}[i]} \text{ and } D(v_q, p) < \bar{l}(\phi)\};$ 
4    $\mathbb{P}_{i+1} \leftarrow \{p | p \in \mathbb{P}_{c_{S_q}[i+1]} \text{ and } D(v_q, p) < \bar{l}(\phi)\};$ 
5    $dist[u] = \inf$  for all  $u \in \mathbb{V} \cup \mathbb{P}$ ,  $dist[p] = 0$  for all  $p \in \mathbb{P}_i$ ;
6   priority_queue  $Q \leftarrow \{p\} \in \mathbb{P}_i$ ;
7   while  $Q$  is not empty do
8      $u \leftarrow Q.dequeue$ ;
9     if  $u$  is already visited then continue;
10    if  $u \in \mathbb{P}_{i+1}$  then
11       $l_s[i] = dist[u]$ ;
12      break;
13    for each  $u'$  for  $e(u, u') \in E$  do
14      if  $dist[u] + w(u, u') < dist[u']$  then
15         $dist[u'] = dist[u] + w(u, u')$ ;
16         $Q.enqueue(u')$ ;
17 return  $l_s$ ;
18 end procedure
```

same as the former executions of the modified Dijkstra algorithms. Thus, we reuse the result starting at the same PoI vertex by materializing the result of the modified Dijkstra algorithm (i.e., keeping PoI vertices matching c_d and distances from p_d to the PoI vertices), which we refer to as *on-the-fly caching*.

After finding SkySRs, on-the-fly caching frees the results of the modified Dijkstra algorithms (this is why we call it *on-the-fly*), because the search space rarely overlaps across different inputs (i.e., S_q and v_q differ).

5.4 Theoretical Analysis

In this section, we theoretically analyze the cost and correctness of the proposed BSSR.

THEOREM 1. (Time complexity) *Let γ be a ratio of pruning and α be a ratio of the size of a graph to find the SkySRs. The time complexity of BSSR is $O(\gamma(\alpha|\mathbb{P}|)^{|S_q|} \alpha(|\mathbb{E}| + (|\mathbb{V}| + |\mathbb{P}|) \log(\alpha(|\mathbb{V}| + |\mathbb{P}|))))$.*

proof: The time complexity of the Dijkstra algorithm is $O(|\mathbb{E}| + |\mathbb{V}| \log |\mathbb{V}|)$ if the number of vertices is $|\mathbb{V}|$. In our setting, we have $|\mathbb{V}| + |\mathbb{P}|$ vertices because we have two types of vertices. In addition, we do not need to search the whole graph by reducing the graph size according to the threshold. Therefore, the time complexity of the modified Dijkstra algorithm is $O(\alpha(|\mathbb{E}| + (|\mathbb{V}| + |\mathbb{P}|) \log(\alpha(|\mathbb{V}| + |\mathbb{P}|))))$. The time complexity of BSSR depends on the number of times the modified Dijkstra algorithms is executed. The number of modified Dijkstra algorithms is equal to all the potential routes $|\mathbb{P}|^{|S_q|}$. Recall that we can prune the number of routes using the branch-and-bound algorithm. Finally, the time complexity of BSSR is $O(\gamma(\alpha|\mathbb{P}|)^{|S_q|} \alpha(|\mathbb{E}| + (|\mathbb{V}| + |\mathbb{P}|) \log(\alpha(|\mathbb{V}| + |\mathbb{P}|))))$. \square

In our approach, γ and α depend on the upper and lower bounds. These are affected by the graph structure, the category trees, and the ratio of PoI vertices, and the time complexity of BSSR depends on these factors.

THEOREM 2. (Space complexity) *Let γ be the pruning ratio, and α be the ratio of the size of the graph to find the SkySRs. The space complexity of BSSR is $O(|\mathbb{E}| + |\mathbb{V}| + |\mathbb{P}| + \gamma|S_q|(\alpha|\mathbb{P}|)^{|S_q|})$.*

proof: We store the whole graph of size $O(|\mathbb{E}| + |\mathbb{V}| + |\mathbb{P}|)$. We also store routes into the priority queue and \mathcal{S} , and the maximum

number of routes is $|\mathbb{P}|^{|S_q|}$. We can prune the number of routes using the branch-and-bound algorithm. The size of the routes is proportional to $|S_q|$. Therefore, the space complexity of BSSR is $O(|\mathbb{E}| + |\mathbb{V}| + |\mathbb{P}| + \gamma|S_q|(\alpha|\mathbb{P}|)^{|S_q|})$. \square

If the number of routes in the priority queue is small, the graph size becomes the main factor related to the memory usage. Otherwise, the number of routes in the priority queue is the main factor.

THEOREM 3. (Correctness) *BSSR guarantees the exact result.*

proof: BSSR prunes routes based on the upper and lower bounds. BSSR safely prunes routes dominated by or equivalent to the obtained sequenced routes. As a result, BSSR does not sacrifice the exactness of the search result. \square

5.5 Running Example

We demonstrate BSSR with optimization techniques using Example 1.1. Table 4 shows routes in priority queue Q_b and sequenced routes in \mathcal{S} . To compute category similarity and semantic score, we use Equations (6) and (7), respectively.

First, we process NNinit, and \mathcal{S} initially includes $\{\langle p_2, p_5, p_8 \rangle, \langle p_2, p_5, p_7 \rangle\}$. 1st step: BSSR starts to find PoI vertices that semantically match Asian restaurant from v_q with the threshold of 15. Then, it finds p_1, p_2, p_6, p_{10} , and p_{11} . Both p_2 's and p_{10} 's category similarities are 1, and their lengths are 6 and 8, respectively. Thus, p_2 comes the top in Q_b . 2nd step: BSSR searches PoI vertices that semantically match Arts&Entertainment from p_2 , and finds p_5 . Since $\langle p_2, p_{12} \rangle$ passes through p_5 and $l(\langle p_2, p_9 \rangle)$ is more than 15, both routes are not inserted to Q_b . 3rd step: as the top route is $\langle p_2, p_5 \rangle$, BSSR searches PoI vertices that semantically match gift shop from p_5 . BSSR does not find any routes due to the threshold. 4th step: BSSR fetches $\langle p_{10} \rangle$ from Q_b and inserts two routes $\langle p_{10}, p_5 \rangle$ and $\langle p_{10}, p_{12} \rangle$ to Q_b . 5th step: BSSR fetches $\langle p_{10}, p_{12} \rangle$ and finds sequenced route $\langle p_{10}, p_{12}, p_{13} \rangle$. Since $\langle p_{10}, p_{12}, p_{13} \rangle$ dominates $\langle p_2, p_5, p_8 \rangle, \langle p_2, p_5, p_8 \rangle$ is deleted from \mathcal{S} . 6th step: The top route $\langle p_{10}, p_5 \rangle$ is deleted from Q_b because its length score is not smaller than the threshold of 13. 7th step: BSSR fetches $\langle p_1 \rangle$ and inserts $\langle p_1, p_5 \rangle$ and $\langle p_1, p_9 \rangle$. 8th step: BSSR fetches $\langle p_1, p_9 \rangle$ and finds a sequenced route $\langle p_1, p_9, p_8 \rangle$. $\langle p_1, p_9, p_8 \rangle$ is inserted to \mathcal{S} , and $\langle p_2, p_5, p_7 \rangle$ is deleted from \mathcal{S} . 9th step: $\langle p_1, p_5 \rangle$ is deleted due to the threshold. 10th step: BSSR fetches $\langle p_6 \rangle$ and finds a route $\langle p_6, p_9 \rangle$. 11th step: BSSR finds a sequenced route $\langle p_6, p_9, p_8 \rangle$, and the route dominates $\langle p_1, p_9, p_8 \rangle$. 12th step: The distance from p_{11} to the PoI vertices that match A&E is larger than the threshold. Finally, BSSR returns the set of SkySRs \mathcal{S} .

6 VARIATIONS AND EXTENSIONS

The SkySR query has a number of variations and extensions. We discuss some of these in the following.

Directed graphs: The SkySR query can be easily applied to directed graphs. We only need to use the Dijkstra algorithm for directed graphs. Here, no modification of the main idea is required.

PoI with multiple categories: To treat PoIs with multiple categories, we can change the definitions of sequenced routes and category similarity. Specifically, we change condition (ii) in Definition 3.4 to state that at least one $c_{p_i}[j]$ ($1 \leq j \leq k_i$) semantically matches $c_S[i]$ for $1 \leq i \leq |S|$, where $c_{p_i}[j]$ is the j -th category of p_i and k_i is the number of categories associated with p_i . The category similarity is either the highest or the average value among the category similarities.

Table 4: Example of BSSR algorithm

0	$Q_b:$ $S: \langle p_2, p_5, p_8 \rangle, \langle p_2, p_5, p_7 \rangle$
1	$Q_b: \langle p_2 \rangle, \langle p_{10} \rangle, \langle p_1 \rangle, \langle p_6 \rangle, \langle p_{11} \rangle$ $S: \langle p_2, p_5, p_8 \rangle, \langle p_2, p_5, p_7 \rangle$
2	$Q_b: \langle p_2, p_5 \rangle, \langle p_{10} \rangle, \langle p_1 \rangle, \langle p_6 \rangle, \langle p_{11} \rangle$ $S: \langle p_2, p_5, p_8 \rangle, \langle p_2, p_5, p_7 \rangle$
3	$Q_b: \langle p_{10} \rangle, \langle p_1 \rangle, \langle p_6 \rangle, \langle p_{11} \rangle$ $S: \langle p_2, p_5, p_8 \rangle, \langle p_2, p_5, p_7 \rangle$
4	$Q_b: \langle p_{10}, p_{12} \rangle, \langle p_{10}, p_5 \rangle, \langle p_1 \rangle, \langle p_6 \rangle, \langle p_{11} \rangle$ $S: \langle p_2, p_5, p_8 \rangle, \langle p_2, p_5, p_7 \rangle$
5	$Q_b: \langle p_{10}, p_5 \rangle, \langle p_1 \rangle, \langle p_6 \rangle, \langle p_{11} \rangle$ $S: \langle p_{10}, p_{12}, p_{13} \rangle, \langle p_2, p_5, p_7 \rangle$
6	$Q_b: \langle p_1 \rangle, \langle p_6 \rangle, \langle p_{11} \rangle$ $S: \langle p_{10}, p_{12}, p_{13} \rangle, \langle p_2, p_5, p_7 \rangle$
7	$Q_b: \langle p_1, p_9 \rangle, \langle p_1, p_5 \rangle, \langle p_6 \rangle, \langle p_{11} \rangle$ $S: \langle p_{10}, p_{12}, p_{13} \rangle, \langle p_2, p_5, p_7 \rangle$
8	$Q_b: \langle p_1, p_5 \rangle, \langle p_6 \rangle, \langle p_{11} \rangle$ $S: \langle p_{10}, p_{12}, p_{13} \rangle, \langle p_1, p_9, p_8 \rangle$
9	$Q_b: \langle p_6 \rangle, \langle p_{11} \rangle$ $S: \langle p_{10}, p_{12}, p_{13} \rangle, \langle p_1, p_9, p_8 \rangle$
10	$Q_b: \langle p_6, p_9 \rangle, \langle p_{11} \rangle$ $S: \langle p_{10}, p_{12}, p_{13} \rangle, \langle p_1, p_9, p_8 \rangle$
11	$Q_b: \langle p_{11} \rangle$ $S: \langle p_{10}, p_{12}, p_{13} \rangle, \langle p_6, p_9, p_8 \rangle$
12	$Q_b:$ $S: \langle p_{10}, p_{12}, p_{13} \rangle, \langle p_6, p_9, p_8 \rangle$

Complex category requirement: We can specify more detailed category requirements, such as *conjunction*, *disjunction*, and *negation*. For example, we can specify that a PoI category is “American restaurant” or “Mexican restaurant” (disjunction), but not “Taco Place” (negation). If PoI vertices are associated with more than two categories, we can specify a conjunction such as “Cafe” and “Bakery”. Note that the time complexity of our algorithm does not change if we specify a detailed requirement because the detailed requirements are equivalent to increasing the number of categories.

Skyline trip planning query: The proposed algorithm can be applied to the trip planning query without category order. For searching routes without category order, the proposed algorithm searches PoI vertices that semantically match a category in a given set of categories. Then, if the algorithm finds PoI vertices, it deletes the categories that are already included in the routes to find next PoI vertices. Note that we need to modify some definition and scoring functions for routes without category order. By this procedure, we can find skyline routes efficiently.

SkySR with destination: Note that we can specify the destination. The simple way to calculate a SkySR with a destination is to add the distance from the last visited PoI vertex to the destination to the length score after finding the sequenced route. To improve efficiency, we traverse PoI vertices from both the destination and the start point.

7 EXPERIMENTAL STUDY

We perform experiments to evaluate the effectiveness of the proposed algorithm. All algorithms are implemented in C++ and run on an Intel(R) Xeon(R) CPU E5620 @ 2.40GHz with 32 GB of RAM.

7.1 Experimental settings

Algorithm. We compare the proposed BSSR and algorithms that iteratively find OSRs using the Dijkstra-based solution and the PNE approach (denoted Dij and PNE, respectively), as described in Section 3. We evaluate performance with respect to (i) response time, and (ii) maximum resident set size (RSS) to represent memory usage.

Table 5: Summary of dataset

Dataset	Area	$ V $	$ E $	$ C $
Tokyo	Tokyo	401,893	174,421	499,397
NYC	New York city	1,150,744	451,051	1,722,350
Cal	California	21,048	87,365	108,863

Dataset. We conduct experiments using various maps (Tokyo, New York city, and California). Table 5 summarizes each dataset. For the Tokyo and NYC datasets, the road network is extracted from OpenStreetMap³ and the PoI information is extracted from Foursquare. Each PoI is embedded on the closest edge in the same way as [10] and is associated with the Foursquare category trees. Note that the number of category trees in Foursquare is 10. For the Cal dataset, the road network and PoI information are available online⁴. The number of categories in the Cal dataset is 63⁵. For each dataset, we use distances based on longitude and latitude as edge weights and treat the graphs as undirected graphs. The graphs are implemented using adjacency lists.

For each dataset, we generate 100 searches, in which the size of a sequence is $|S_q|$. The start points are selected randomly from vertices in the maps. The categories of sequences are selected randomly from the leaf nodes in the category trees with the constraint that they have different category trees. Since the number of PoI vertices associated with each category is significantly biased, we select only categories that have a large number of PoI vertices.

Here, category similarity is calculated based on the *Wu and Palmer similarity measure* [19] and the semantic score is calculated as the product of the category similarities of the sequence members. Specifically, we calculate the category similarity and semantic score using the following equations:

$$\text{sim}(c, c') = \max_{c_i \in a(c')} \frac{2 \cdot d(c_m)}{d(c) + d(c')}, \quad (6)$$

$$s(\mathbf{R}) = 1 - \prod_{i=1}^{\min(|R|, |S_q|)} \text{sim}(c_{PR[i]}, c_{S_q}[i]), \quad (7)$$

where $a(c)$, $d(c)$, and c_m denote the set of ancestor categories of c (including c), the depth of c , and the deepest common ancestor category of c and c_i , respectively.

7.2 Overview of results

First, we present an overview of the performance of all algorithms. Figure 3 shows the response time with various category sequence sizes, and Table 6 shows the RSS for a category sequence of size four. Here, “BSSR w/o Opt” denotes BSSR without optimization techniques. In Figure 3, there are missing bars for the case of size of sequence 5, because the executions were not finished after a month.

BSSR achieves the least response time with all datasets and reduces the search space by exploiting the branch-and-bound algorithm and the proposed optimization techniques. By comparing BSSR and BSSR w/o Opt, we confirm that the optimization techniques increase efficiency. When the size of the category sequence is small, PNE finds SkySRs efficiently because it can search for sequenced routes efficiently if the category sequence size is small. On the other hand, as category sequence size increases, the response time of PNE and Dij increases significantly.

³<https://www.openstreetmap.org>

⁴<http://www.cs.utah.edu/~lifeifei/SpatialDataset.htm>

⁵Since the PoIs in the Cal dataset have no category tree information, we generate a category of height three where a non-leaf node has three child nodes.

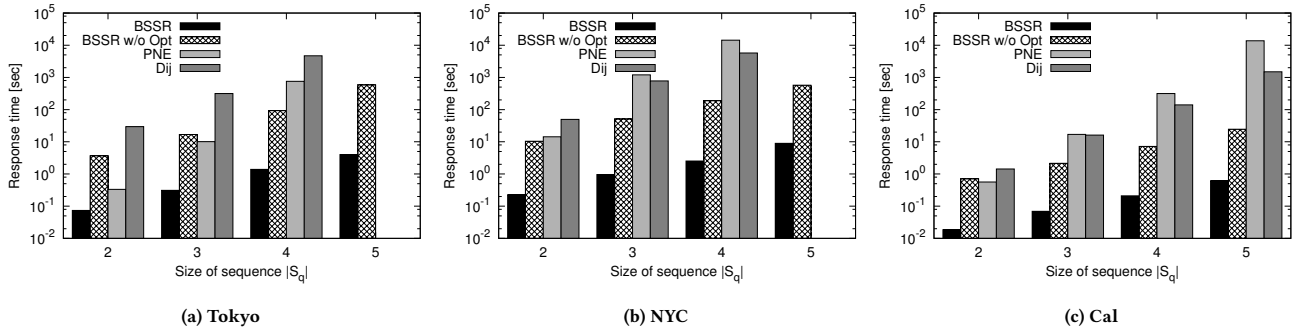


Figure 3: Results obtained for the datasets with various $|S_q|$

Table 6: RSS Comparison

	BSSR	BSSR w/o Opt	PNE	Dij
Tokyo	239.6 MB	497.5 MB	239.8 MB	4.8 GB
NYC	658.0 MB	659.4 MB	658.7 MB	9.7 GB
Cal	36.7 MB	53.7 MB	36.6 MB	70.3 MB

Table 7: Effect of initial search for various $|S_q|$

Dataset	Approach	Metrics	2	3	4	5
Tokyo	Proposed	Weight sum	0.009	0.013	0.017	0.021
		Response time [msec]	3.5	5.1	6.9	8.6
		# of routes	1.49	1.33	1.36	1.49
		Ratio	0.74	0.79	0.82	0.86
Existing	Weight sum	0.32 (regardless $ S_q $)				
NYC	Proposed	Weight sum	0.044	0.066	0.073	0.078
		Response time [msec]	10.7	16.5	19.5	24.1
		# of routes	1.76	1.79	1.81	1.82
		Ratio	0.67	0.81	0.85	0.83
Existing	Weight sum	1.31 (regardless $ S_q $)				
Cal	Proposed	Weight sum	0.79	1.28	1.57	1.85
		Response time [msec]	1.4	2.3	2.9	3.9
		# of routes	2.27	2.37	2.28	2.25
		Ratio	0.70	0.79	0.85	0.86
Existing	Weight sum	12.14 (regardless $ S_q $)				

If the category sequence size is large, BSSR achieves better performance than PNE and Dij even if we do not use optimization techniques. By comparing Dij to PNE, it can be seen that their performance depends on the datasets and the category sequence size. Although the PNE approach was proposed to be a more sophisticated algorithm than the Dijkstra-based solution [16], PNE requires more time than Dij for the NYC and Cal datasets, which implies that it is not effectively robust to datasets. In terms of RSS, BSSR and PNE achieve nearly the same performance. These two algorithms do not store many routes in the priority queue; therefore, RSS is highly dependent on the graph size. On the other hand, as Dij stores many routes in the priority queue, RSS is significantly larger than those of the other algorithms. Although we do not show the routes returned by each algorithm due to space limitations, all algorithms output the same routes. As a result, BSSR achieves the fastest response time with small memory usage without sacrificing the exactness of the result.

7.3 Optimization Techniques

The optimization techniques improve the efficiency of BSSR. Here, we evaluate each optimization technique.

Initial Search: We show the search spaces with and without an initial search for the first modified Dijkstra algorithm to evaluate the effect of the initial search. Moreover, we evaluate NNinit in terms of response time. Table 7 shows the weight sum, which

Table 8: Effect of priority queue for various $|S_q|$

Dataset	Approach	2	3	4	5
Tokyo	Proposed	3750	17600	112000	397000
	Distance-based	3890	23500	189000	1760000
NYC	Proposed	13800	108000	172000	637000
	Distance-based	14800	165000	444000	1520000
Cal	Proposed	4900	24800	84900	383000
	Distance-based	5300	34900	168000	899000

represents the search space, the response time of NNinit, and the number of sequenced routes found by NNinit for various category sequence sizes. In addition, we show the ratio of the length score of the sequenced route with the largest semantic score among the sequenced routes found in the initial search to the length score of the sequenced route whose semantic score is 0 in the initial search. The weight sum with the initial search is significantly smaller than that without the initial search. We can avoid traversing the whole graph using the initial search; thus, this can significantly reduce the search space of BSSR. Moreover, since the response time of NNinit is significantly less than that of BSSR (Figure 3), we confirm that NNinit can reduce the search space efficiently. Note that the number of sequenced routes found by the initial search is not large. On the other hand, the length score of the sequenced route with the largest semantic score is much smaller than that of the sequenced route whose semantic score is 0. As a result, NNinit reduces the search space significantly without increasing total response time.

Tightening Upper Bound: The priority queue aims at efficiently tightening the upper bound to reduce the search space. Here, we show the total number of vertices visited by BSSR, which is highly related to the response time. Table 8 shows the total number of vertices visited by the proposed priority queue and distance-based priority queue for various category sequence sizes. The number of vertices visited by the proposed priority queue is less than that of the distance-based priority queue. In particular, as the size of the category sequences increases, the performance gap increases because, as the category sequence size increases, the distance-based priority queue cannot find sequenced routes efficiently. Thus, the upper bound is rarely updated. On the other hand, the proposed priority queue can update the upper bound efficiently because the route with the largest size is dequeued preferentially. Thus, the proposed priority queue is more suitable than the distance-based approach for finding SkySRs.

Tightening Lower Bound: To tighten the lower bound, we propose two types of possible minimum distances, i.e., semantic-match and perfect-match minimum distances. If the minimum possible distance is large, we can prune routes even if the routes

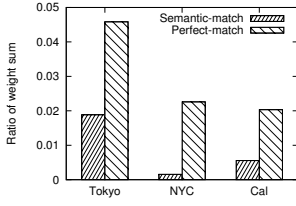
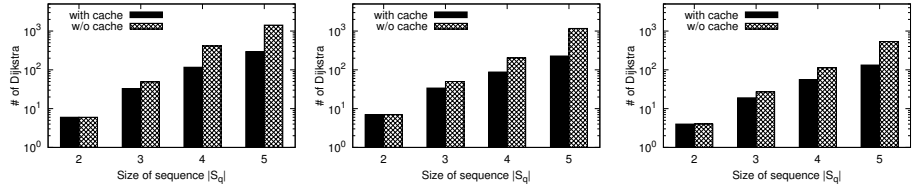


Figure 4: Effect of minimum possible distances



(a) Tokyo

(b) NYC

(c) Cal

Figure 5: Effect of on-the-fly caching for various $|S_q|$

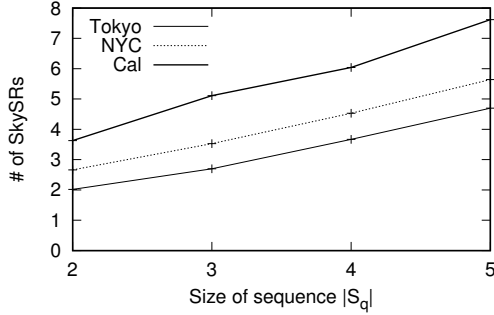


Figure 6: Number of SkySRs for various $|S_q|$

include a small number of PoI vertices. Figure 4 shows the ratios of the possible minimum distances to the sum weights of the initial search when we set the category sequence size to five. The semantic-match and perfect-match minimum distances in the Tokyo dataset effectively reduce the search space by tightening the lower bound. However, different from the Tokyo dataset, the possible minimum distances in the NYC and Cal datasets are small. Since the PoI vertices in the two datasets are relatively concentrated in a small area, the possible minimum distances become small. The effect of the possible minimum distances highly depends on the skews of locations of the PoI vertices.

On-the-fly Caching: On-the-fly caching can reuse the results of former modified Dijkstra algorithm executions; thus, the number of executions of the Dijkstra algorithm decreases. Figure 5 shows the numbers of executions of modified Dijkstra algorithms by BSSR with all optimization techniques and those except for on-the-fly caching. The number of executions of the Dijkstra algorithms decreases using on-the-fly caching. In particular, when the category sequence size increases, the performance gap increases because, as the category sequence size increases, we have more opportunities to reuse former results. Thus, we confirm that on-the-fly caching is effective to reduce the number of executions of the Dijkstra algorithms.

7.4 Number of skyline sequenced routes

Figure 6 shows the number of SkySRs obtained with each dataset for various $|S_q|$. As shown, the Cal dataset returns the largest number of SkySRs. The response time and RSS obtained with the Tokyo and NYC datasets are much greater than those of the Cal dataset, which implies that the number of SkySRs does not affect response time and RSS significantly. Moreover, if we use a complete real-world dataset, we may not require a ranking function because the number of SkySRs would be small.

Table 9: Example SkySRs in Tokyo

Distance	Sequenced route
7451 meters	Beer Garden → Sushi Restaurant → Sake Bar
1295 meters	Bar → Sushi Restaurant → Sake Bar

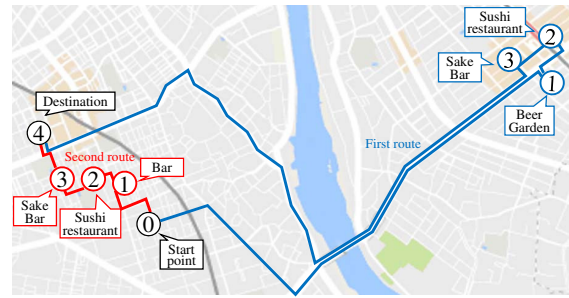


Figure 7: Visualization of routes in Tokyo: black circles (with 0 and 4) denote a start point and a destination, respectively. Blue and red circles denote sequences of PoIs for the first and second routes in Table 9, respectively, and their numbers indicate the order of PoIs to be visited.

7.5 Usecase

We show an example of SkySRs in Tokyo. We assume that we plan to go to places for dinner and drinks. We want to visit a “Beer garden”, a “Sushi restaurant”, and a “Sake bar” from our current location and finally go to our hotel. Table 9 and Figure 7 show two representative SkySRs from the four identified SkySRs. Note that the other two routes are similar to either of the representative routes. In the Foursquare category trees, “Bar” includes “Beer Garden” and “Sake bar”, and “Japanese restaurant” includes “Sushi restaurant”. Thus, we find routes using “Bar” and/or “Japanese restaurant”. The second route is much shorter than the first route that perfectly matches the user requirement, and the difference between them is only whether they pass a “Bar” or “Beer garden”. The best route depends on the users and situations (e.g., weather); thus, we confirm that SkySRs are useful to help users make decisions.

8 USER STUDY

We developed a prototype SkySR query service⁶ using OpenStreetMap and the Santander Open Data platform from Santander, Spain⁷. Figure 8 shows a screenshot of the prototype system, which outputs one of the SkySR route. We performed a test in July, 2017. To gather users for this test, the Santander municipality arranged meetings with different groups of people

⁶<https://ss.festival.ckp.jp/OuRouteSuggestion/dispSearchRoute/index>. The default language is Spanish.

⁷<http://datos.santander.es>

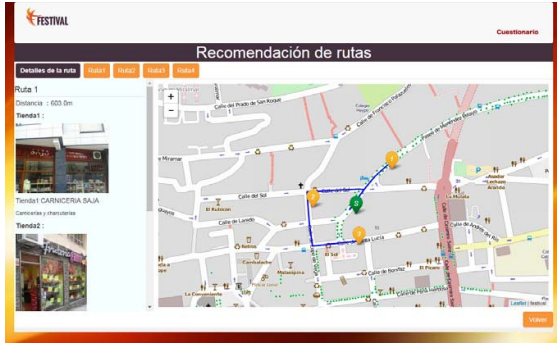


Figure 8: Screenshot of the prototype system

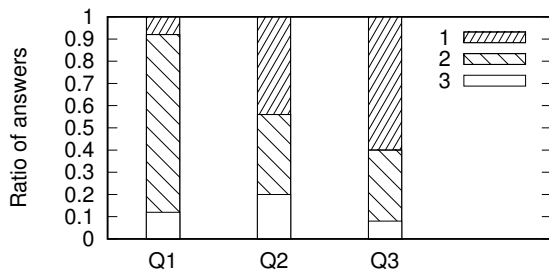


Figure 9: Ratios of answers for each question

to present the service: municipal staff (computing, convention and tourism municipal services), students from vocational training departments who are developing webpages and apps, and citizens. We also provided a leaflet that shows the concept of the SkySR query and how to use the service. In this test, users freely used the service and answered a questionnaire (25 respondents). The questionnaire included the following three questions.

Q1 What do you think about this service?

Answer. 1. I love it, 2. I like it, 3. I do not like it.

Q2 Would you recommend it to anyone?

Answer. 1. Yes, 2. Maybe, 3. No.

Q3 Do you think that it is a good idea for the city: citizens, tourists, commercial sectors?

Answer. 1. Yes, 2. Maybe, 3. No.

We summarize the ratios of answers for each question in Figure 9. As shown, more than 80% of the users liked the service. In addition, the questionnaire shows that the service is valuable for the city. From the user experiment, we confirm that the SkySR query is useful for users and cities.

9 CONCLUSION

In this paper, we have first introduced a semantic hierarchy for trip planning. We then proposed the skyline sequenced route (SkySR) query, which finds all preferred routes from a start point according to a user's PoI requirements. In addition, we have proposed an efficient algorithm for the SkySR query, i.e., BSSR, which simultaneously searches for all SkySRs by a single traversal of a given graph. To optimize the performance of BSSR, we proposed four optimization techniques. We evaluated the proposed approach using real-world datasets and demonstrated that it comprehensively outperforms naive approaches in terms of response time without increasing memory usage or sacrificing the exactness of the result. Moreover, we developed a SkySR

query service using open data, and conducted a user test, which confirmed that SkySR queries are useful for both users and cities.

In future work, we would like to extend the proposed approach in several directions. First, because we assume a forest structure for the category classification in this paper, a more complex classification may provide better granularity. Second, because we have not used any preprocessing techniques such as indexing, we plan to propose a suitable preprocessing method for the SkySR query. Finally, although the SkySR query proposed in this paper considers two scores (length and category similarity), it could be extended to consider many attributes of a PoI (e.g., text, keywords, and ratings) and the cost/quality of a graph (e.g., route popularity, tolls, and the number of traffic lights).

ACKNOWLEDGEMENT

This research is partially supported by the Grant-in-Aid for Scientific Research (A)(JP16H01722) and Grant-in-Aid for Young Scientists (B)(JP15K21069).

REFERENCES

- [1] Saad Aljubayrin, Zhen He, and Rui Zhang. 2015. Skyline Trips of Multiple POIs Categories. In *DASFAA*. 189–206.
- [2] S Börzsöny, Donald Kossmann, and Konrad Stocker. 2001. The Skyline Operator. In *ICDE*. 421–430.
- [3] Haiquan Chen, Wei-Shinn Ku, Min-Te Sun, and Roger Zimmermann. 2008. The Multi-rule Partial Sequenced Route Query. In *ACM SIGSPATIAL GIS*. 1–10.
- [4] Jian Dai, Chengfei Liu, Jiajie Xu, and Zhiming Ding. 2016. On Personalized and Sequenced Route Planning. *World Wide Web* 19, 4 (2016), 679–705.
- [5] Jochen Eisner and Stefan Funke. 2012. Sequenced route queries: Getting things done on the way back home. In *ACM SIGSPATIAL*. 502–505.
- [6] Pierre Hansen. 1980. Bicriterion path problems. In *Multiple criteria decision making theory and application*. 109–127.
- [7] Xuegang Huang and Christian S Jensen. 2005. In-route skyline querying for location-based services. In *W2GIS*. 120–135.
- [8] H-P Kriegel, Matthias Renz, and Matthias Schubert. 2010. Route Skyline Queries: A Multi-preference Path Planning Approach. In *ICDE*. 261–272.
- [9] Eugene L Lawler and David E Wood. 1966. Branch-and-bound Methods: A Survey. *Operations research* 14, 4 (1966), 699–719.
- [10] Feifei Li, Dihan Cheng, Marios Hadjieleftheriou, George Kollios, and Shang-Hua Teng. 2005. On Trip Planning Queries in Spatial Databases. In *SSTD*. 273–290.
- [11] Jing Li, Yin David Yang, and Nikos Mamoulis. 2013. Optimal Route Queries with Arbitrary Order Constraints. *TKDE* 25, 5 (2013), 1097–1110.
- [12] Xiaobin Ma, Shashi Shekhar, Hui Xiong, and Pusheng Zhang. 2006. Exploiting a Page-level Upper Bound for Multi-type Nearest Neighbor Queries. In *ACM GIS*. 179–186.
- [13] Ernesto Queiros Vieira Martins. 1984. On a multicriteria shortest path problem. *European Journal of Operational Research* 16, 2 (1984), 236–245.
- [14] Yutaka Ohsawa, Htoo Htoo, Noboru Sonehara, and Masao Sakauchi. 2012. Sequenced Route Query in Road Network Distance based on Incremental Euclidean Restriction. In *DEXA*. 484–491.
- [15] Philip Resnik. 1995. Using Information Content to Evaluate Semantic Similarity in a Taxonomy. In *IJCAI*. 448–453.
- [16] Mehdi Sharifzadeh, Mohammad Kolahdouzan, and Cyrus Shahabi. 2008. The Optimal Sequenced Route Query. *The VLDB Journal* 17, 4 (2008), 765–787.
- [17] Michael Shekelyan, Gregor Jossé, and Matthias Schubert. 2015. Linear Path Skylines in Multicriteria Networks. In *ICDE*. 459–470.
- [18] Yuan Tian, Ken CK Lee, and Wang-Chien Lee. 2009. Finding Skyline Paths in Road Networks. In *ACM SIGSPATIAL GIS*. 444–447.
- [19] Zhibiao Wu and Martha Palmer. 1994. Verbs Semantics and Lexical Selection. In *ACL*. 133–138.
- [20] Bin Yang, Chenjuan Guo, Christian S Jensen, Manohar Kaul, and Shuo Shang. 2014. Stochastic Skyline Route Planning under Time-varying Uncertainty. In *ICDE*. 136–147.

On Complexity and Efficiency of Mutual Information Estimation on Static and Dynamic Data

Michael Vollmer
 Karlsruhe Institute of Technology
 Karlsruhe, Germany
 michael.vollmer@kit.edu

Ignaz Rutter*
 Eindhoven University of Technology
 Eindhoven, The Netherlands
 i.rutter@tue.nl

Klemens Böhm
 Karlsruhe Institute of Technology
 Karlsruhe, Germany
 klemens.boehm@kit.edu

ABSTRACT

Mutual Information (MI) is an established measure for the dependence of two variables and is often used as a generalization of correlation measures. Existing methods to estimate MI focus on static data. However, dynamic data is ubiquitous as well, and MI estimates on it are useful for stream mining and advanced monitoring tasks. In dynamic data, small changes (e.g., insertion or deletion of a value) may often invalidate the previous estimate. In this article, we study how to efficiently adjust an existing MI estimate when such a change occurs. As a first step, we focus on the well-known nearest-neighbor based estimators for static data and derive a tight lower bound for their computational complexity, which is unknown so far. We then propose two dynamic data structures that can update existing estimates asymptotically faster than any approach that computes the estimates independently, i.e., from scratch. Next, we infer a lower bound for the computational complexity of such updates, irrespective of the data structure and the algorithm, and present an algorithm that is only a logarithmic factor slower than this bound. In absolute numbers, these solutions offer fast and accurate estimates of MI on dynamic data as well.

1 INTRODUCTION

Motivation. Finding and quantifying dependencies between variables is an essential task in data analysis. Conventional methods to detect (in)dependent attributes, like correlation coefficients and covariance matrices, are limited in the types of dependencies they detect. *Mutual Information* (MI) in turn is a notion from Information Theory that captures both linear and arbitrary non-linear dependencies. However, MI is defined on the probability density of the data. This makes exact computation impossible on samples. Nevertheless, existing MI estimators yield good results even for small samples [13]. In consequence, a wide range of applications, such as Feature Selection [22], Text Analysis [7] and Computer Vision [23], uses MI.

A popular choice are estimators based on nearest-neighbor distances [9, 16, 17]. This is because such estimators essentially are non-parametric and yield very good results [13, 14, 21, 29]. Nearest-neighbor based estimation of MI is often perceived as equivalent to the concrete estimation formula by Kraskov et al. (KSG)[17]. However, the KSG is just one estimation formula for MI using the nearest-neighbor entropy estimator by Kozachenko and Leonenko [16]. There exists at least one other MI estimator using a different formula, while relying on the same entropy estimator (3KL)[9]. In the following, the term *estimator* names

concrete formulas that estimate the MI value (e.g., KSG, 3KL), and *nearest-neighbor based estimation* is the group of these estimators.

So far, algorithms to compute nearest-neighbor based MI estimates and thus their practical applications focus on static data. However, data streams are ubiquitous as well and also require suitable analysis methods. The problem studied in this article is nearest-neighbor based estimation of MI on dynamic data. In this setting, an elementary task called *update* is the incremental maintenance of an estimate when adding or deleting a point.

With dynamic data, scalability with the number of data points is crucial. A good, data-independent measure is the computational complexity of the respective algorithms. In order to evaluate the efficiency of a new solution, it also is important to know the complexity of the problem. That is, a lower bound for any algorithm that computes such estimates, independent of the concrete approach. So far, no lower bound for nearest-neighbor based MI estimation is known, be it for updating an existing estimate, be it for computing the estimate on static data.

Challenges. Designing the estimators envisioned with controlled complexity is challenging. Two reasons for this are as follows: First, while nearest-neighbor based estimation of entropy depends on distances to the nearest neighbor, this does not imply that it has the same computational complexity as nearest-neighbor search. Put differently, it may be possible to obtain the same results using different methodologies. Consequently, one must prove the complexity based only on the result and not hinge on the complexity of certain tasks that seem mandatory in the context at first sight, such as nearest-neighbor search.

Second, to design a dynamic data structure that answers certain queries faster than any static algorithm, it is necessary to identify expensive computations whose results are relatively easy to maintain in the presence of updates. This means that the time required to incrementally maintain the results after a change must be limited in all cases. But this is not obvious here. At the same time, availability of these results must significantly speed up the query.

Our Contributions. Our work focuses on the time required to maintain an estimate of MI on dynamic data. We concentrate on the computational efficiency of nearest-neighbor based estimators on static data and the implications for dynamic estimation. We present solutions for dynamic data that maintain an estimate with the same estimation quality as static estimators, but with less time required. Specifically, our contributions are as follows:

Computational complexity of nearest-neighbor based estimators. In Section 4 we provide a complexity analysis of nearest-neighbor based MI estimators. Using a proof by reduction, we establish a lower bound in the algebraic computation tree model for any algorithm estimating MI using 3KL or KSG. To our knowledge, we are the first to prove any lower bound for the time complexity of such estimators. The lower bounds we prove are tight. This means that there already exist algorithms that have this asymptotically

*This work originated while the author was affiliated with Karlsruhe Institute of Technology.

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

optimal running time. Additionally, we use this result to infer lower bounds for the maintenance of KSG and 3KL estimates on dynamic data.

Dynamic data structures. In this article, we present two dynamic data structures. The first one is DEMI, which estimates nearest-neighbor based MI on a dynamic data set, see Section 5. This data structure holds a set of data points and some intermediate computation results we use for the estimation. The data structure allows insertion and deletion of data points and querying the estimate using all data points stored. Both the 3KL and the KSG estimator can use this data structure. We prove that updating an estimate using DEMI is asymptotically faster than the lower bound for static estimates, i.e., computing the estimate from scratch. To our knowledge, we are first to present a way of maintaining a KSG estimate on dynamic data that requires asymptotically less time than static estimation and preserves the estimate without any approximation.

Near-optimal computation time. The second data structure we present, ADEMI, integrates existing state-of-the-art data structures and algorithms into DEMI to reduce computation time, see Section 6. While the structure does not offer a speedup when maintaining the KSG estimate, we can maintain the 3KL estimate in polylogarithmic time. In particular, we are only a logarithmic factor slower than the lower bound shown in Section 4.

Systematic experimental evaluation. Finally, we evaluate our approaches experimentally, using a broad variety of dependency types and noise levels, in Section 7. We show that both KSG and 3KL converge to the true MI values with a good rate of convergence. This is a stark contrast to another recently published estimator for MI on sliding windows [3]. Additionally, we show that our data structures perform very well when maintaining MI estimates on large samples.

2 FUNDAMENTALS

We begin by revisiting the foundations of MI and its estimation.

Mutual Information. Introduced by Shannon [26], the notion of *entropy* is a measure for the expected information from observing the value of a random variable X , noted as $H(X)$. The expected information for observed values of two random variables X and Y is the natural extension *joint entropy* $H(X, Y)$. This gives way to the notion of *Mutual Information*

$$I(X; Y) = H(X) + H(Y) - H(X, Y), \quad (1)$$

which describes the information shared between both variables. Using the definition of entropy for continuous random variables in Equation 1 yields the differential definition of MI

$$I(X; Y) = \int_Y \int_X p(x, y) \log \left(\frac{p(x, y)}{p(x)p(y)} \right) dx dy \quad (2)$$

where $p(x)$, $p(y)$ and $p(x, y)$ are the marginal and joint probability density functions of X and Y , respectively [8]. Using the natural logarithm, MI is then measured in the *natural unit of information* (nat).

Nearest-Neighbor based Estimation. Kozachenko and Leonenko [16] presented a nearest-neighbor based estimator for (joint) entropies for a given sample. They used the distance to the k -th nearest neighbor as a means to approximate the density of the distribution for that region. They also have proven that this method yields a consistent estimator for entropy independent of the choice of k . ‘Consistent’ means that, with increasing sample

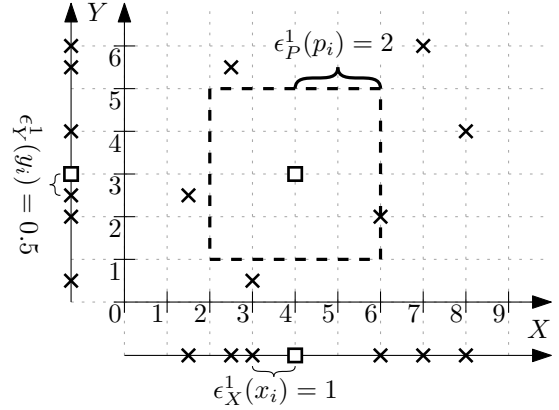


Figure 1: Illustration of the notation used for the 3KL.

size, the estimate converges towards the true entropy value. Their method is as follows:

Let $Q = \{q_1, \dots, q_n\} \subseteq \mathbb{R}^d$ be a set of points in a d -dimensional euclidean space, and let $\epsilon_Q^k(q_i)$ be the distance between q_i and its k -th nearest neighbor in Q using the L_∞ -norm, also known as maximum distance. Using the notation by Kraskov et al. [17], the entropy estimator by Kozachenko and Leonenko [16] is

$$\hat{H}(Q) = \psi(n) - \psi(k) + \log(2^d) + \frac{d}{n} \sum_{i=1}^n \log(\epsilon_Q^k(q_i)), \quad (3)$$

where ψ is the digamma function. That is, $\psi(x) = \sum_{m=1}^{x-1} (\frac{1}{m}) - C$, for $x \geq 1$ with $C \approx 0.577$ being the Euler-Mascheroni-Constant.

We now say how this entropy estimator is used to estimate MI. Let $P = \{p_1 = (x_1, y_1), \dots, p_n = (x_n, y_n)\} \subseteq \mathbb{R}^2$ be a sample of a random variable with two attributes. Note that we use P for the sample whose MI value we are interested in. This may be the original, full data set as well as the set of the most recent points of a data stream or any other subsample. Additionally, let $X = \{x_1, \dots, x_n\}$ and $Y = \{y_1, \dots, y_n\}$ be the sets of all values of the respective attribute in the sample. We use $\epsilon_P^k(p_i)$, $\epsilon_X^k(x_i)$ and $\epsilon_Y^k(y_i)$ to refer to the distance of p_i , x_i and y_i to its k -th nearest neighbor in P , X and Y , respectively. Figure 1 illustrates an exemplary set P , with $k = 1$ and p_i , x_i and y_i marked as squares. Inserting Equation 3 into Equation 1 yields the MI estimator

$$\widehat{I_{3KL}}(P) = \psi(n) - \psi(k) + \frac{1}{n} \sum_{i=1}^n \log \left(\frac{\epsilon_X^k(x_i) \cdot \epsilon_Y^k(y_i)}{(\epsilon_P^k(p_i))^2} \right). \quad (4)$$

Because this estimator estimates each of the three entropies in Equation 1 separately with the estimator by Kozachenko and Leonenko, we call the estimator 3KL. This estimator has also been used by Evans [9] for MI estimation on static data. Additionally, because each term in Equation 1 is estimated using a consistent estimator, the 3KL also is a consistent estimator.

Varying numbers of nearest neighbors. A different approach to use Equation 3 for MI estimation was proposed by Kraskov et al. [17]. While the 3KL uses the same k when estimating $\hat{H}(X)$, $\hat{H}(Y)$ and $\hat{H}(P)$ to obtain a compact formula, Kraskov et al. adjust k for every point such that the logarithmic term is 0. The idea is to make the distances $\epsilon_P^k(p_i)$, $\epsilon_X^k(x_i)$ and $\epsilon_Y^k(y_i)$ of a point to its nearest neighbors in X , Y and P identical. To achieve this, the parameter k for $\epsilon_X^k(x_i)$ and $\epsilon_Y^k(y_i)$ has to be set accordingly and may be different for each point. Specifically, each nearest neighbor p_j of p_i in P should result in x_j being a nearest neighbor

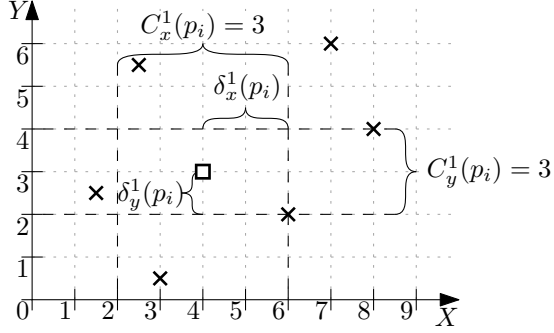


Figure 2: Illustration of the notation used for the KSG.

of x_i in X and y_j being a nearest neighbor of y_i in Y . To this end, the k -th nearest neighbor distance $\epsilon_P^k(p_i)$ is determined and afterwards k_x and k_y for $\epsilon_X^{k_x}(x_i)$ and $\epsilon_Y^{k_y}(y_i)$ is set accordingly. Figure 2 features an illustration of the notation that follows, using the same exemplary set P and k as Figure 1. As before, the set kNN of k nearest neighbors of a point $p_i \in P$ using the L_∞ -norm is determined first. Let $\delta_x^k(p_i) = \max_{p_j \in kNN} |x_i - x_j|$ be the greatest distance between x_i and any other x -value among its k nearest neighbors. Then the *marginal count* $C_x^k(p_i)$ is the number of elements in X as close to x_i as this distance, i.e.,

$$C_x^k(p_i) = |\{x \in X \setminus \{x_i\} : |x_i - x| \leq \delta_x^k(p_i)\}|. \quad (5)$$

Another marginal count $C_y^k(p_i)$ is defined analogously using y_i and $\delta_y^k(p_i)$. When these marginal counts are used as k per point in the estimator $\hat{H}(X)$ and $\hat{H}(Y)$, the distances $\epsilon_X(x_i)$, $\epsilon_Y(y_i)$ and $\epsilon_P(p_i)$ in Equation 4 (mostly) cancel out. However, one has to adjust the formula for using a different k , i.e., the marginal counts, for each point. The resulting estimator, called KSG due to its inventors Kraskov, Stögbauer and Grassberger, is

$$\widehat{I}_{KSG}(P) = \psi(n) + \psi(k) - \frac{1}{k} - \frac{1}{n} \sum_{i=1}^n \psi(C_x^k(p_i)) + \psi(C_y^k(p_i)). \quad (6)$$

While there exist many other approaches to estimate MI, we focus on nearest-neighbor based estimation due to its performance in comparative studies [13, 14, 21, 29]. Because these studies consider only the KSG, we use the same distributions in Section 7 to assess the estimation quality of the 3KL.

Another point is that it is generally recommended [13, 14, 17] to use a small k , that is $k < 10$. The choice of k has only been studied extensively for the KSG but not for the 3KL. However, because Equation 3 is consistent for any k [16], the 3KL is the sum of three consistent estimators and thus consistent for any k as well. Consequently, we assume $k < 10$ in this work, i.e., k is a constant for asymptotic considerations.

3 RELATED WORK

Data Streams. Data streams, a constantly growing form of dynamic data, are ubiquitous. Because data streams grow over time, and memory and storage is limited, it is impossible to store all data points. This means that information is lost over time.

Nevertheless, there exist space-efficient estimators for entropy of discrete distributions on streams. With Equation 1, entropy estimators can also be used to estimate MI, but with accumulating error. The estimator of Chakrabarti et al. [5] provides multiplicative approximations of entropy on insert-only streams. In contrast, the estimator by Harvey et al. [10] offers multiplicative and

additive approximations of entropy on streams with insertions and deletions, but requires knowledge about the maximum length of the stream. However, both estimators are restricted to discrete distributions. Estimating MI on discrete distributions is easier, because their relative count of points is a good estimator for the probability. Estimating the density of continuous distributions in turn is not trivial.

The estimation of MI of continuous distributions on streams has received less attention. The MISE framework [12] offers estimates of MI between continuous variables for any time interval on data streams. While both MISE and our approach offer nearest-neighbor based MI estimation, the difference is as follows: Results with MISE are approximations of the KSG estimate for consecutive subsets of the data. We in turn provide exactly the estimates of KSG and 3KL on a dynamic data set. Maintaining an accurate KSG estimate for a dynamic set of data points, e.g., the last 1000 data points, would incur prohibitively high (and growing) resource consumption with MISE. This is because it cannot explicitly delete points. In consequence, the target application and the optimizations are too different to allow for a fair comparison.

Sliding Windows. A common approach to process data streams are sliding windows. Maintaining only a fixed number of points ensures a fixed problem size that allows for bounded resource consumption. By construction, this technique rules out the usage of any information outside the window, but allows for accurate computations on data within it. There already are very good general approaches for sliding-window aggregation [27]. However, no competitive MI estimator is known so far that can be aggregated and thus used with this framework. Most MI estimators have stronger relations to concrete items than to collective values, e.g., distances to the nearest neighbor instead of distances to the mean. In consequence, previous analytics tasks that use MI estimates over a sliding window [15, 24] had to recompute the estimate from scratch for each window.

There is little work regarding algorithmic optimization of the computation time for such tasks. A very recent work by Boidol and Hapfelmeier [3] has introduced an estimator that approximates the 3KL inside a sliding window. In contrast, our approach allows for arbitrary insertions and deletions, and we provide the exact results of the 3KL and KSG. To show the difference between their approximation and accurate 3KL estimates, we include their method in our experiments in Section 7.

Computational Complexity. There has been little research regarding the computational complexity of the KSG and 3KL. Several proposals to compute the KSG appear in the original KSG article [17] with the claimed time complexity $O(n)$ for their fastest, so-called “box-assisted” algorithm on smooth distributions. Vajmelka et al. [28] compare their own approach with the box-assisted algorithm and cite [25] for different conditions for a linear runtime of the box-assisted algorithm. In the end, the best universal time complexity of their presented algorithms is $\Theta(n \log n)$. The same complexity is given for the algorithm computing the 3KL by Evans [9]. In the following section we prove that this limit is not a coincidence, i.e., we prove that no algorithm computing these estimators can have a time complexity lower than $O(n \log n)$.

4 LOWER BOUNDS

In this section we present our first contribution, the lower bounds for computing and maintaining estimates using the KSG and 3KL.

All existing approaches to compute the 3KL and KSG follow the original description in the sense that they first compute the nearest neighbors of all points. In the case of the KSG, the marginal counts C_x^k and C_y^k are computed afterwards. However, it is not known if this is the only approach to compute $\widehat{I_{3KL}}(P)$ and $\widehat{I_{KSG}}(P)$, or if it is computationally optimal. For instance, there could be a different formula for either of these estimators that does not require explicit computation of the nearest neighbors. Consequently, the complexity of computing the 3KL and KSG can only be based on the result and not on intermediate steps such as determining the nearest neighbors. The problems whose complexities we want to study in general, i.e., without confinement to specific algorithms, are the following ones.

Problem 1 (3KL-ESTIMATION). For a set $P \subseteq \mathbb{R}^2$ of points, determine $\widehat{I_{3KL}}(P)$.

Problem 2 (KSG-ESTIMATION). For a set $P \subseteq \mathbb{R}^2$ of points, determine $\widehat{I_{KSG}}(P)$.

In the following, we show the complexity of Problem 1. By reducing a problem with known complexity to 3KL-ESTIMATION, we prove that it has a lower bound of $\Omega(n \log n)$ in the algebraic computation tree model [1]. For brevity, all formal proofs in this article are available in Appendix A. We use the algebraic computation tree model because it allows us to prove bounds without assuming any statistical properties of the data. This is important because we want general-purpose estimation of MI. If knowledge regarding the data or its distribution was known, it could be used to model the density function in Equation 2.

THEOREM 4.1. *The problem 3KL-ESTIMATION has time complexity $\Omega(n \log n)$.*

PROOF. The formal proof is available in Appendix A.1. \square

This lower bound matches the running time of the algorithm presented by Evans [9] to solve 3KL-ESTIMATION. Consequently, this algorithm is already asymptotically optimal, and the lower bound is tight.

COROLLARY 4.2. *The computational complexity of 3KL-ESTIMATION is $\Theta(n \log n)$.*

We use the same approach to prove a lower bound for KSG-ESTIMATION. With the algorithms presented by Vejmelka et al. [28] this lower bound is tight as well.

THEOREM 4.3. *The problem KSG-ESTIMATION has a time complexity in $\Omega(n \log n)$.*

PROOF. The formal proof is available in Appendix A.2. \square

COROLLARY 4.4. *The computational complexity of KSG-ESTIMATION is $\Theta(n \log n)$.*

As a next step, we consider dynamic data. The distinctive feature of dynamic data is that the data changes over time. For a set P of points, all changes can be modeled using insertion of new points and deletion of existing points. For instance, moving a point from (x, y) to (x', y') can be modeled with one deletion of (x, y) and one insertion of (x', y') . To maintain an estimate of MI with the 3KL or KSG, we need to adjust the estimate according to such insertions or deletions. We see this as a problem for a dynamic data structure and thus allow storage of some auxiliary information about P , noted as *state* S_P of a dynamic data structure. The formal problem is then:

Problem 3 (3KL-UPDATE). Let $P \subseteq \mathbb{R}^2$ be a set of points, S_P the state for P and $p \in \mathbb{R}^2$ a point. Determine $\widehat{I_{3KL}}(P \cup \{p\})$ and $S_{P \cup \{p\}}$ if p is inserted and $\widehat{I_{3KL}}(P \setminus \{p\})$ and $S_{P \setminus \{p\}}$ if p is deleted using only S_P and p .

Problem 4 (KSG-UPDATE). Let $P \subseteq \mathbb{R}^2$ be a set of points, S_P the state for P and $p \in \mathbb{R}^2$ a point. Determine $\widehat{I_{KSG}}(P \cup \{p\})$ and $S_{P \cup \{p\}}$ if p is inserted and $\widehat{I_{KSG}}(P \setminus \{p\})$ and $S_{P \setminus \{p\}}$ if p is deleted using only S_P and p .

Because these problems can be used to solve 3KL-ESTIMATION and KSG-ESTIMATION, respectively, we can use the previous results to infer lower bounds for their time complexities. If we start with an empty set P and incrementally insert n points, the total time required cannot generally be asymptotically faster than $\Omega(n \log n)$ by Theorem 4.1 and Theorem 4.3. Because this includes n insertions, the time complexity of individual insertions is in $\Omega(\log n)$.

COROLLARY 4.5. *The problem 3KL-UPDATE has a time complexity in $\Omega(\log n)$.*

COROLLARY 4.6. *The problem KSG-UPDATE has a time complexity in $\Omega(\log n)$.*

In this section we have established formal problem descriptions for the tasks of estimating and maintaining MI estimates using the 3KL and KSG. Furthermore, we have proven lower bounds for the time required to solve these problems. These bounds are tight for computing estimates on static data. This means that no asymptotic speed-up is achievable. In contrast, we are not aware of any data structures or algorithms that solve the problems of maintaining 3KL or KSG estimates when points are inserted or deleted. In the following sections, we present two data structures for these tasks, evaluate their time complexity and compare them to the lower bounds presented in this section.

5 ESTIMATING MUTUAL INFORMATION ON DYNAMIC DATA

Naturally, the simplest solution to KSG-UPDATE and 3KL-UPDATE is storing exactly P in S_P and computing $\widehat{I_{3KL}}(\cdot)$ and $\widehat{I_{KSG}}(\cdot)$, respectively, with every change. The result from the previous section is that any such approach would require $\Omega(n \log n)$ time for 3KL-UPDATE and KSG-UPDATE. In the following we show that this is not optimal and present a more efficient solution.

We propose the data structure DEMI (Dynamic Estimation of Mutual Information) that focuses on updating an estimate of the 3KL or KSG for a single insertion or deletion. First, we present how this data structure works with 3KL estimates. In Section 5.2 we describe the differences when maintaining a KSG estimate. In more detail, we describe the changes to the 3KL estimate that can occur by inserting or deleting a point. Then we describe which information our data structure stores and how it determines the changes in the 3KL estimate efficiently. Lastly, we evaluate the space complexity of our data structure as well as the time complexity of adding or deleting a point.

5.1 Updating 3KL Estimates

Let $P = \{p_1 = (x_1, y_1), \dots, p_n = (x_n, y_n)\} \subseteq \mathbb{R}^2$ be the set of points in our sample and let $X = \{x_1, \dots, x_n\}$ and $Y = \{y_1, \dots, y_n\}$ be the set of values per attribute. When we insert a point $p_{n+1} = (x_{n+1}, y_{n+1}) \in \mathbb{R}^2$, let $P' = P \cup \{p_{n+1}\}$, $X' = X \cup \{x_{n+1}\}$ and $Y' = Y \cup \{y_{n+1}\}$ be the sets including

Data Structure 1: DEMI

```
struct {  
  | real  $x, y$   
  | real  $\epsilon_p^k, \epsilon_X^k, \epsilon_Y^k$   
} DemiPoint;  
  
struct {  
  | DemiPoint[ ]  $P_D$   
  | BST<DemiPoint*>  $T_x, T_y$   
  | real  $base, sum$   
} state;
```

p_{n+1}, x_{n+1} and y_{n+1} , respectively. Considering Equation 4, the change from $\widehat{I_{3KL}}(P)$ to $\widehat{I_{3KL}}(P')$ consists of three partial changes:

- (1) $\psi(n)$ increases to $\psi(n+1) = \psi(n) + \frac{1}{n}$,
- (2) the arithmetic mean includes $n+1$ logarithms instead of n ,
- (3) and the nearest-neighbor distances $\epsilon_p^k(p_i)$, $\epsilon_X^k(x_i)$ and $\epsilon_Y^k(y_i)$ may change for any $i \in \{1, \dots, n\}$.

While Change (1) is trivial, Change (2) requires the computation of $\epsilon_p^k(p_{n+1})$, $\epsilon_X^k(x_{n+1})$ and $\epsilon_Y^k(y_{n+1})$. However, Change (3) could require the re-evaluation of all nearest-neighbor distances. Clearly, these changes apply analogously if p_1 is removed from P instead of inserting p_{n+1} . Following these observations, we propose a dynamic data structure that determines these changes efficiently and evaluate its computation complexity.

Overview. Our data structure, DEMI, is given in Data Structure 1. For each point $p_i \in P$ of our sample, we store its attributes x_i, y_i and k -th nearest-neighbor distances $\epsilon_p^k(p_i)$, $\epsilon_X^k(x_i)$ and $\epsilon_Y^k(y_i)$ as a *DemiPoint*. In addition, we store references to all DemiPoints, ordered by the x -component and y -component of the point, in binary search trees (BST) T_x and T_y , respectively. Using self-balancing BST like red-black-trees, we can insert, delete and search items in logarithmic time. Additionally, we also maintain the values $base = \psi(|P|) - \psi(k)$ and $sum = \sum_{i=1}^n \log \left(\frac{\epsilon_X^k(x_i) \cdot \epsilon_Y^k(y_i)}{(\epsilon_p^k(p_i))^2} \right)$. The collection of all stored data is the state S_P of our data structure for the sample P . Because we store a constant amount of information per point, the space complexity of DEMI is $\Theta(n)$. Given State S_P , one can query the 3KL estimate on the set P in constant time as $\widehat{I_{3KL}} = base + \frac{sum}{|P|}$. However, this data structure requires adjustment of S_P after every change of P .

Insertion Algorithm. To insert a point p_{n+1} into a state S_P , illustrated in Algorithm 2, we distinguish two phases of the update. First (Lines 1-6), we add p_{n+1} as a DemiPoint to P_D and update $base$ and sum accordingly. Second (Lines 7-18), we determine which nearest-neighbor distances change and adjust sum according to the changes. We now describe these steps in more detail, together with the computational complexity of elementary operations, to allow for an easier evaluation. We discuss possible improvements in Section 6.

To add p_{n+1} to S_P , we first compute its k -th nearest neighbor in P' by linear search and derive the k -th nearest-neighbor distance $\epsilon_p^k(p_{n+1})$ ($O(n)$, Line 1). To determine the k -th nearest neighbor distances $\epsilon_X^k(x_{n+1})$ and $\epsilon_Y^k(y_{n+1})$ we can use the binary search tree and evaluate the distance to the next k and preceding k elements ($O(k \cdot \log n)$, Line 2). With this information we construct the DemiPoint for p_{n+1} and insert it into P_D ($O(1)$,

Algorithm 2: INSERT(S_P, p_{n+1})

```
1 Compute  $\epsilon_p^k(p_{n+1})$   $O(n)$   
2 Compute  $\epsilon_X^k(x_{n+1})$  and  $\epsilon_Y^k(y_{n+1})$   $O(k \cdot \log n)$   
3 Insert  $p_{n+1}$  into  $P_D$   $O(1)$   
4 Reference  $p_{n+1}$  in  $T_x, T_y$   $O(\log n)$   
5  $base \leftarrow base + \frac{1}{n}$   $O(1)$   
6  $sum \leftarrow sum + \log \left( \frac{\epsilon_X^k(x_{n+1}) \cdot \epsilon_Y^k(y_{n+1})}{(\epsilon_p^k(p_{n+1}))^2} \right)$   $O(1)$   
7  $A \leftarrow \{p_i \in P: \max(|x_i - x_{n+1}|, |y_i - y_{n+1}|) < \epsilon_p^k(p_i)\}$   $O(n)$   
8  $B \leftarrow \{p_i \in P: |x_i - x_{n+1}| < \epsilon_X^k(x_i)\}$   $O(n)$   
9  $C \leftarrow \{p_i \in P: |y_i - y_{n+1}| < \epsilon_Y^k(y_i)\}$   $O(n)$   
10 forall  $p_i \in A$  do  
11   | Compute  $\epsilon_p^k(p_i)$   $O(|A| \cdot n)$   
12   |  $sum \leftarrow sum + \log((\epsilon_p^k(p_i))^2) - \log((\epsilon_p^k(p_i))^2)$   $O(|A|)$   
13 forall  $p_i \in B$  do  
14   | Compute  $\epsilon_X^k(x_i)$   $O(|B| \cdot k \cdot \log n)$   
15   |  $sum \leftarrow sum - \log(\epsilon_X^k(x_i)) + \log(\epsilon_X^k(x_i))$   $O(|B|)$   
16 forall  $p_i \in C$  do  
17   | Compute  $\epsilon_Y^k(y_i)$   $O(|C| \cdot k \cdot \log n)$   
18   |  $sum \leftarrow sum - \log(\epsilon_Y^k(y_i)) + \log(\epsilon_Y^k(y_i))$   $O(|C|)$ 
```

Line 3). References to this point are then inserted into T_x and T_y ($O(\log n)$, Line 4). Then, we add the appropriate terms to $base$ and sum ($O(1)$, Lines 6 and 7), respectively.

Next, we find all previous nearest-neighbor distances that changed, by linear search. For each $i \in \{1, \dots, n\}$ we test whether p_{n+1}, x_{n+1} and y_{n+1} is closer than $\epsilon_p^k(p_i)$, $\epsilon_X^k(x_i)$ and $\epsilon_Y^k(y_i)$, respectively. This takes time in $O(n)$ and yields the sets A, B and C (Lines 7-9), respectively. For each point $p_i \in A$ we compute $\epsilon_p^k(p_i)$ analogously to $\epsilon_p^k(p_{n+1})$, which takes $O(n)$ each. Then we adjust sum accordingly ($O(1)$, Line 12). The sets A and B are handled in an analogous way, using $\epsilon_X^k(x_i)$ and $\epsilon_Y^k(y_i)$, respectively, instead (Lines 13-18). Note that these distances can be computed in time $O(k \cdot \log n)$ each, instead of $O(n)$, analogous to $\epsilon_X^k(x_{n+1})$ and $\epsilon_Y^k(y_{n+1})$.

Computational Complexity. The total runtime for inserting a point into our structure therefore is in $O(k \cdot n + |A| \cdot n + (|B| + |C|) \cdot k \cdot \log n)$. In the following theorem we show that $|A|, |B|$ and $|C|$ are in $O(k)$, because there are at most $8 \cdot k$ points for which p_{n+1} is one of the k nearest neighbors. Consequently, our insertion time is in $O(k \cdot n + k^2 \cdot \log n)$. Since k is suggested to be a small constant, e.g. less than 10, in the literature, we can assume k to be constant. This means that an insertion is in $O(n)$. This results in the total time complexity of $O(n)$. Because deleting a point changes the estimate analogously, we can use an analogous algorithm with the same complexity, i.e., $O(n)$.

THEOREM 5.1. *Let $P \subseteq \mathbb{R}^2$ be a set of points. For any point $p \in P$ there exist at most $8k$ points $q \in P$ such that p is one of the k nearest neighbors of q using the L_∞ -norm.*

PROOF. The formal proof is available in Appendix A.3. \square

As context for the update time of $O(n)$, Theorem 4.1 proves that any algorithm requires time in $\Omega(n \log n)$ to compute the 3KL from scratch. As a result, updating an estimate using DEMI is asymptotically faster than recomputing it, independently of the method used. In Section 6 we show how the time for updates

Data Structure 3: DEMI-KSG

```
struct {  
  real  $x, y$   
  real  $\epsilon_p^k, \delta_x^k, \delta_y^k$   
  int  $C_x^k, C_y^k$   
} DemiPointKSG;  
  
struct {  
  DemiPointKSG[ ]  $P_D$   
  BST<DemiPointKSG* >  $T_x, T_y$   
  real  $base, sum$   
} state;
```

on the 3KL can be improved even further. However, we will first discuss how we use the same approach to update KSG estimates.

5.2 Updating KSG Estimates

In this section, we describe how we achieve the same results, that is linear space and linear time for updates, using KSG estimates instead of 3KL estimates. As with the 3KL, we decompose the KSG estimate into $\widehat{I_{KSG}} = base + \frac{sum}{|P|}$. Comparing Equation 4 and Equation 6, it follows that $base$ and sum need to maintain different values when maintaining 3KL or KSG estimates. The change for $base$, that is $base = \psi(|P|) + \psi(k) - \frac{1}{k}$ instead of $base = \psi(|P|) - \psi(k)$, does not have any influence on the overall procedure. However, the change from $sum = \sum_{i=1}^n \log \left(\frac{\epsilon_x^k(x_i) \cdot \epsilon_y^k(y_i)}{(\epsilon_p^k(p_i))^2} \right)$ to $sum = -\sum_{i=1}^n \psi(C_x^k(p_i)) + \psi(C_y^k(p_i))$ has stronger implications. Most notably, we do not require explicit nearest neighbor distances per point but need marginal counts. We need to update a marginal count $C_x^k(p_i)$ if and only if the nearest neighbors of p_i in P changes, or a point (x, y) with $|x - x_i| \leq \delta_x^k(p_i)$ is inserted or deleted, see Figure 2. As a consequence, per point p_i we do not store $\epsilon_x^k(x_i)$ and $\epsilon_y^k(y_i)$ but the distances to the furthest x - and y -values among the k nearest neighbors in P , i.e., $\delta_x^k(p_i)$ and $\delta_y^k(p_i)$. Additionally we track the marginal counts $C_x^k(p_i)$ and $C_y^k(p_i)$. These slight changes are displayed in Data Structure 3. Furthermore, this means that we still store a constant amount of information per point, and the space complexity of the data structure remains $\Theta(n)$.

Updating the data structure follows the same principles as before, that is, we include the new point into the data structure and evaluate its impact on other marginal counts afterwards. In the following we describe the changes in specific steps between the update algorithm for 3KL estimates and KSG estimates, that is, Algorithm 2 and Algorithm 4.

Tracking marginal counts, instead of nearest-neighbor distances, per attribute allows for faster updates, because the counts only need increments and decrements ($O(1)$ each, Lines 16 and 18), instead of recomputation. However, a change of nearest neighbors does also invalidate the marginal counts and requires computing them and correct adjustment of sum (Lines 11-14). Computing marginal counts from scratch can be done with linear search ($O(n)$ each, Lines 2 and 13).

Regarding the time complexity of Algorithm 4, it is important to note that B and C are not sets of points with changed nearest neighbors. As a consequence, only the size of A has an upper bound of $8 \cdot k$ by Theorem 5.1. In the worst case, B and C contain all points, that is, $|B| \leq n$ and $|C| \leq n$. The total time complexity

Algorithm 4: INSERT-KSG(S_P, p_{n+1})

```
1 Compute  $\delta_x^k(p_{n+1}), \delta_y^k(p_{n+1})$  and  $\epsilon_{p'}^k(p_{n+1})$   $O(n)$   
2 Compute  $C_x^k(p_{n+1})$  and  $C_y^k(p_{n+1})$   $O(n)$   
3 Insert  $p_{n+1}$  into  $P_D$   $O(1)$   
4 Reference  $p_{n+1}$  in  $T_x, T_y$   $O(\log n)$   
5  $base \leftarrow base + \frac{1}{n}$   $O(1)$   
6  $sum \leftarrow sum - \psi(C_x^k(p_{n+1})) - \psi(C_y^k(p_{n+1}))$   $O(1)$   
7  $A \leftarrow \{p_i \in P: \max(|x_i - x_{n+1}|, |y_i - y_{n+1}|) < \epsilon_{p'}^k(p_i)\}$   $O(n)$   
8  $B \leftarrow \{p_i \in P: |x_i - x_{n+1}| < \delta_x^k(p_i)\}$   $O(n)$   
9  $C \leftarrow \{p_i \in P: |y_i - y_{n+1}| < \delta_y^k(p_i)\}$   $O(n)$   
10 forall  $p_i \in A$  do  
11    $sum \leftarrow sum + \psi(C_x^k(p_i)) + \psi(C_y^k(p_i))$   $O(|A|)$   
12   Compute  $\delta_x^k(p_i), \delta_y^k(p_i)$  and  $\epsilon_{p'}^k(p_i)$   $O(|A| \cdot n)$   
13   Compute  $C_x^k(p_i)$  and  $C_y^k(p_i)$   $O(|A| \cdot n)$   
14    $sum \leftarrow sum - \psi(C_x^k(p_i)) - \psi(C_y^k(p_i))$   $O(|A|)$   
15 forall  $p_i \in B$  do  
16    $sum \leftarrow sum - \frac{1}{C_x^k(p_i)}$ ;  $C_x^k(p_i) \leftarrow C_x^k(p_i) + 1$   $O(|B|)$   
17 forall  $p_i \in C$  do  
18    $sum \leftarrow sum - \frac{1}{C_y^k(p_i)}$ ;  $C_y^k(p_i) \leftarrow C_y^k(p_i) + 1$   $O(|C|)$ 
```

therefore is $O(n + |A| \cdot n) = O(k \cdot n)$. As before, k is taken as constant, which yields the time complexity $O(n)$. This is asymptotically faster than recomputing the estimate by Theorem 4.3.

6 POLYLOGARITHMIC UPDATES

Because DEMI relies only on simple algorithms like linear search and binary search trees during insertions and deletions, faster solutions might exist. In this section we determine which parts of our insertion algorithm have a high computational cost and present solutions for these tasks. There are two factors that lead to the linear time complexity of Algorithm 2.

- (1) Computing the nearest neighbors, with linear search
- (2) Finding the points whose nearest neighbors changed by linear search

6.1 Geometric Structures

Computing the nearest neighbors. Computing the k nearest neighbors of a point is a classic problem of computational geometry, which has received a lot of research. While there exist many solutions, most of them are built for static data and are not compatible with the incremental changes in dynamic data. But there also exist solutions that allow for insertions and deletions. Chan [6] proposed a dynamic data structure that computes nearest neighbors in two-dimensional spaces with sub-linear times for insertion, deletion and queries. However, the computational complexity of deletions is $O(\log^6 n)$, which is quite high. Kapoor and Smid [11] provide an alternative based on dynamic range trees [30]. With dynamic fractional cascading [20] the time complexities for insertions, deletions and querying the nearest neighbor of a point are in $O(\log n \log \log n)$. To query two nearest neighbors, we can query one nearest neighbor, delete this point from the tree, query the new nearest neighbor and insert the deleted point. Querying the k nearest neighbors can thus easily be achieved through a sequence of k queries, $k - 1$ deletions, and $k - 1$ insertions, with total time in $O(k \cdot \log n \log \log n)$.

Finding the points whose nearest neighbors have changed. Finding all points whose nearest neighbors have changed is also a geometric problem, that is, finding the reverse nearest neighbors of the inserted or deleted point. For each point $p = (x, y)$ with nearest neighbor distance ϵ , all nearest neighbors of p (using the L_∞ -norm) are within the square $[x - \epsilon, x + \epsilon] \times [y - \epsilon, y + \epsilon] \subseteq \mathbb{R}^2$. To find all points whose nearest neighbors contain a point p' , the task is to determine which squares contain p' . One data structure to solve this problem is the segment tree by Bentley [2]. The technique of dynamic fractional cascading is also applicable for segment trees [20] and yields the time complexities for insertions and deletions in $O(\log n \log \log n)$. Queries require time in $O(\log n \log \log n + m)$, where m is the number of squares returned.

6.2 Improving DEMI

To achieve sublinear time complexity for updates, we integrate a two-dimensional dynamic range tree and a two-dimensional dynamic segment tree into DEMI. We call this the augmented version of DEMI (ADEMI). The insertion algorithm is nearly identical to Algorithm 2, except for changes in time complexities and insertions and deletions to the integrated tree structures. In consequence, we only mention the changes relative to Algorithm 2 in this section. The full data structure and insert algorithm can be found in Appendix B.

Using the dynamic range tree, Line 1 requires only time in $O(k \cdot \log n \log \log n)$, and Line 11 requires time in $O(|A| \cdot k \cdot \log n \log \log n)$. Using the dynamic segment tree, Line 7 can be done in time $O(\log n \log \log n + |A|)$. Additionally, B and C can only contain elements that are at most k positions before and after x_{n+1} and y_{n+1} in T_x and T_y , respectively. Consequently, Lines 8–9 can also be done using the binary search trees in time $O(k \cdot \log n)$.

Additionally, we need to maintain the integrated tree structures. Specifically, we insert p_{n+1} into the dynamic range tree and insert the square of its nearest neighbors, that is,

$$\begin{aligned} \text{square}(p_{n+1}, P') &= [x_{n+1} - \epsilon_{p'}^k(p_{n+1}), x_{n+1} + \epsilon_{p'}^k(p_{n+1})] \\ &\quad \times [y_{n+1} - \epsilon_{p'}^k(p_{n+1}), y_{n+1} + \epsilon_{p'}^k(p_{n+1})], \end{aligned} \quad (7)$$

into the dynamic segment tree. The dashed lines in Figure 1 illustrate this square. Both insertions require time in $O(\log n \log \log n)$. Finally, for each point $p_i \in A$ we delete its old square of nearest neighbors $\text{square}(p_i, P)$ from the dynamic segment tree and insert the new square $\text{square}(p_i, P')$. This requires time in $O(|A| \cdot \log n \log \log n)$.

For an overview of the new time complexity, the updated insertion algorithm can be found in Appendix B. Because $|A|, |B|, |C| \in O(k)$, the total time complexity of an insertion is $O(k^2 \cdot \log n \cdot \log \log n)$. As before, k can be assumed to be a small constant, which leads to an insertion time of $O(\log n \log \log n)$. Deleting a point is completely analogous to insertions in (A)DEMI, and the used tree structures have the same complexity for insertions and deletions. Consequently, deletions in ADEMI also have a deletion time of $O(\log n \log \log n)$. Since the time complexity of queries is in $O(1)$, ADEMI solves problem 3KL-UPDATE in time $O(\log n \log \log n)$. This means that ADEMI is a nearly optimal, since its time complexity is only a factor $\log \log n$ higher than the lower bound from Corollary 4.6.

The drawback of ADEMI is an increased space complexity. The space complexity of the two-dimensional range tree and segment tree are $O(n \log n)$ and $O(n \log^2 n)$, respectively. Additionally, the improvements to the time complexity cannot be used

when maintaining KSG estimates. This is because the number of points whose marginal counts change during an update has no bound lower than n . Additionally, the impact of incrementing or decrementing a marginal count on the overall estimate depends on the current count, which can be any value between k and n . As a consequence, it remains unclear whether any dynamic data structure can solve KSG-UPDATE in sublinear time, or whether there exists a stronger lower bound.

7 EXPERIMENTS

In this section we empirically validate the estimation quality and time efficiency of our approach. To this end, we use data with known MI values and show that the 3KL converges to these values even with small samples. We also do so for the KSG. For brevity we only present the results for $k = 4$, since this value offers good rates of convergence for both the KSG and 3KL and follows the general recommendation of small values for k . Additionally, we compare the runtimes for maintaining 3KL estimates using ADEMI, DEMI and repeated estimation from scratch (REFS). For REFS we compute Equation 4 repeatedly with a state-of-the-art static approach [9], i.e., using sorting and space-partitioning trees for nearest-neighbor searches. While we have already proven a clear hierarchy regarding their asymptotic scalability, the complexity classes neglect constant factors. So it remains interesting how their concrete runtimes compare.

Setup. All approaches are implemented in C++ and compiled using the Gnu Compiler (v. 5.4) with optimization (-O3) enabled. We use the non-commercial ALGLIB¹ implementation of KD-Trees as space-partitioning trees in REFS. We conduct all experiments on Ubuntu 16.04.2 LTS using a single core of an AMD Opteron™ Processor 6212 clocked at 2.6 GHz and 128GB RAM.

7.1 Data

For our evaluation, we use synthetic and real data sets. In particular, we use the dependent distributions with noise used for comparing MI estimators [13]. These distributions have a noise parameter σ_r , which we vary from 0.1 to 1.0. Thus, we use 10 distributions for each of these dependency types. Additionally, we use the uniform distributions used to compare MI with the maximal information coefficient [14] as well as independent uniform and normal distributions. As real data sets, we use sensor data of randomly charged and discharged batteries [4] and time series of household power consumption [18]. Monitoring MI on such data could be useful to monitor the condition of battery cells for maintenance or to infer knowledge about the behavior of the households inhabitants. In the following, we briefly describe the different distributions and data sets.

Linear. To construct the point $p_i \in P$, we draw the value x_i from the normal distribution $N(0, 1)$. Additionally, we draw some noise r_i from the normal distribution $N(0, \sigma_r)$, where σ_r is the noise parameter of the distribution. This yields the point $p_i = (x_i, x_i + r_i)$.

Quadratic. This distribution is generated analogously to the linear distribution, except that the point is $p_i = (x_i, x_i^2 + r_i)$.

Periodic. For each point $p_i \in P$, we draw the value x_i from the uniform distribution $U[-\pi, \pi]$. Additionally, we draw some noise r_i from the normal distribution $N(0, \sigma_r)$, where σ_r is the noise parameter. This yields the point $p_i = (x_i, \sin(x_i) + r_i)$.

¹ALGLIB (www.alglib.net), Sergey Bochkanov

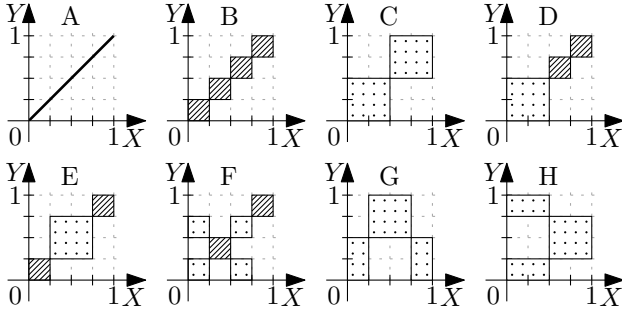


Figure 3: An overview of the uniform distributions used.

Chaotic. This distribution uses the classical Hénon Map, that is,

$$\begin{aligned} h_{x_{i+1}} &= 1 - \alpha \cdot h_{x_i}^2 + h_{y_i} \\ h_{y_{i+1}} &= \beta \cdot h_{x_i}, \end{aligned}$$

with $\alpha = 1.4$, $\beta = 0.3$ and $(h_{x_0}, h_{y_0}) = (0, 0)$. For a point p_i we additionally independently draw noise r_{x_i}, r_{y_i} from the distribution $N(0, \sigma_r)$, where σ_r is the noise parameter. Each point $p_i \in P$ is then $p_i = (h_{x_i} + r_{x_i}, h_{y_i} + r_{y_i})$.

Uniform. The uniform distributions A to H we use are illustrated in Figure 3. Note that the striped areas contain twice as many points as the dotted areas. For these distributions, each striped area with size $0.25 \cdot 0.25$ contains 25% of all points, while dotted areas of the same size contain 12.5% of all points. The distribution A simply draws values v_i from $U[0,1]$ and constructs the points $p_i = (v_i, v_i)$.

Independent. Lastly, we use the distributions U_{IND} and N_{IND} , where each point consists of two values drawn independently and identically distributed from $U[0, 1]$ and $N(0, 1)$, respectively.

Battery Data. This data set, available at the NASA Prognostics Center of Excellence [4], monitors voltage, current and temperature of battery cells during random loads. We use the data corresponding to battery cell “RW9” and use each combination of the attributes as bivariate sample.

Power Consumption. This data set, available at the UCI Machine Learning Repository [18], monitors the power consumption of a household in France. We use each combination of *global active power*, *global reactive power* and *voltage* as a bivariate sample.

Data Precision. The nearest-neighbor based entropy estimator, and by consequence the 3KL and KSG, expects samples from continuous distributions and require samples without duplicate values. Because of the limited precision of the battery and the power consumption data, we add noise to the sample. Kraskov et al. also have observed this issue and recommend the addition of low intensity noise, e.g., a normal distribution with variance 10^{-10} , to eliminate duplicate points [17]. However, we think that filling the missing precision with uniform noise is a better compensation for rounded or imprecise data. Figure 4 illustrates both approaches with the number of duplicates per value of an imprecise data set in parentheses. For our experiments we use the second approach.

7.2 Quality of Estimation

To evaluate the quality of estimation, we use all data sets with well-defined MI values. That is, all synthetic data sets except the chaotic distributions, whose probability densities are unknown,

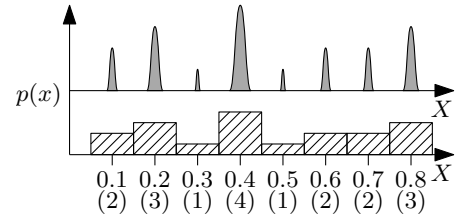


Figure 4: Avoiding duplicates in a sample by adding minimal noise (top) or filling the missing precision uniformly (bottom).

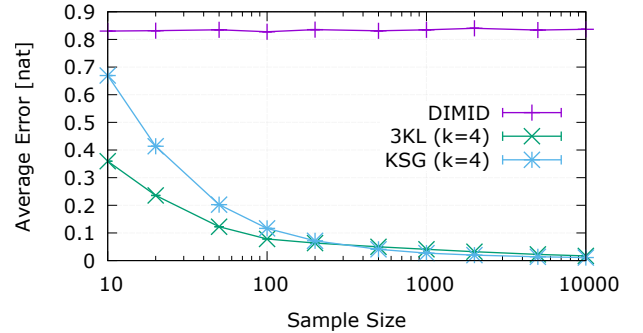


Figure 5: Average difference of estimates to true MI values depending on sample size.

and the uniform distribution A, whose MI is infinite. We use these distributions to evaluate the consistency and the rate of convergence of the KSG, 3KL and the estimator used by DIMID [3]. Specifically, we are interested in the difference between the estimated MI and true value for the distribution as well as the variance of estimates for samples of the same distribution. Since the behavior has turned out to be very homogeneous across the different distributions, we restrict our presentation to selected results.

Development with sample size. For each distribution we created samples with sample sizes between 10 and 10000 and 1000 repeats per size. Figure 5 graphs the average difference between the estimate and the true MI value of the respective distribution. Additionally, Figure 6 shows the standard deviation of estimates of the same distribution and sample size, averaged across all distributions. One can see in these diagrams, that both the 3KL and the KSG converge quickly to the true values and have only small variance. In contrast, the approximate estimator in DIMID has a strong variance and difference. We think the reason is the random projection used by that estimator. It may retain enough information such that estimates are comparable to each other, as shown in their work [3]. However, we think that the projection loses too much information regarding the joint probability density to allow for good MI estimates.

Different dependency types. We also studied whether the quality of estimation changes for different dependency types. As we have seen in the previous paragraph, both the 3KL and KSG are very consistent even with moderate sample sizes. As a result we will use a small sample size, i.e. 100, to highlight differences. Figure 7 shows the average estimation error and standard deviation of estimates using 3KL, KSG and DIMID for each dependency type. While the variance of both KSG and 3KL are comparable

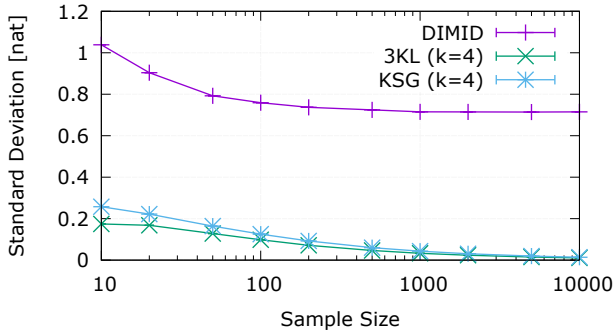


Figure 6: Standard deviation of estimates depending on sample size.

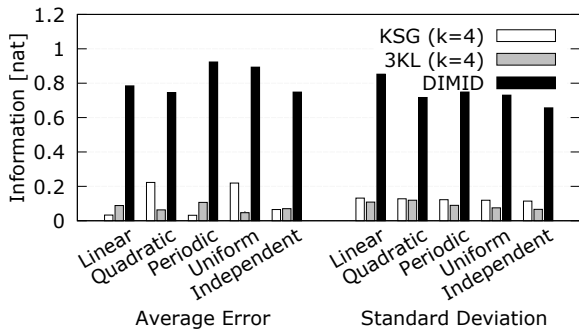


Figure 7: Average difference (left) and standard deviation (right) of estimates to true MI values on distribution type.

for all dependency types, the difference to the true value is imbalanced for the KSG but not the 3KL. Unfortunately, we do not have any explanation for this difference. As before, we notice strong differences between the DIMID approximation and the results of KSG and 3KL.

7.3 Runtime Analysis

We have benchmarked runtimes of our data structures for all data sets. Because we are not aware of any competitor that offers good MI estimates on dynamic data, we compare our performance to naïve recomputation of the estimate when an update occurs. We compare the runtime to maintain 3KL estimates using DEMI and ADEMI as well as repeated recomputation (REFS). We use a slight simplification of the ADEMI trees, compared to the description in Section 6. Specifically, we did not implement dynamic fractional cascading and relied only on the technique of Willard [30] for insertion and deletion of nodes. The reason is that dynamic fractional cascading provides a small asymptotic benefit, i.e. reducing a factor $\log n$ to $\log \log n$, but requires a lot of overhead. As a result, the structure labeled ADEMI in this section has insertion and deletion time in $O(\log^2 n)$ instead of $O(\log n \log \log n)$.

By design, both DEMI and ADEMI require only constant time for querying the current MI value, but require more time to update the data structure during insertions and deletions. The repeated static estimation REFS has inverse properties, i.e., constant time insertions and deletions but expensive queries. To provide a good overview we use the task of monitoring the MI of a changing data set of fixed size. That is, each update consists of deleting one point, inserting a different point and querying

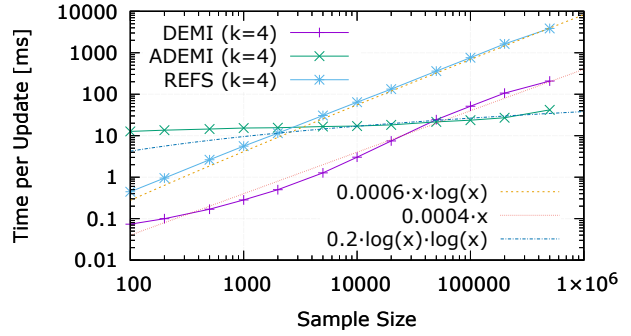


Figure 8: Average time for an update depending on sample size for the synthetic distributions.

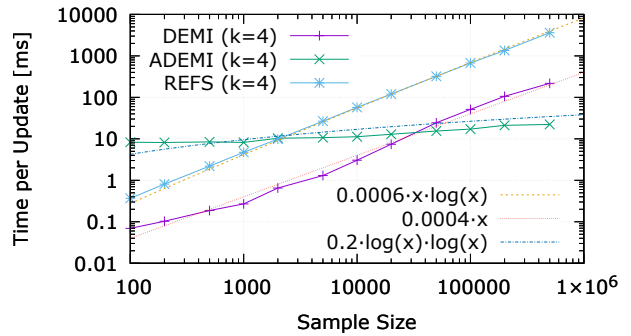


Figure 9: Average time for an update depending on sample size for the used real data set.

the current MI estimate. For these experiments we averaged the time required per update using 1000 updates per distribution and sample size.

Figure 8 shows the average update time required across all synthetic distributions per sample size. The same graph based on the real data sets instead of the synthetic distributions is Figure 9. As expected, the time complexity of each approach translates directly to asymptotic scaling with sample sizes, that is, steepness of the curve in the double log plot. To highlight this, the graphs include different asymptotic functions with dashed and dotted lines. An interesting result is that ADEMI has by far the worst performance for small windows and by far the best performance for large ones. Our explanation is as follows: The maintenance of the range trees and even more so the segment trees is expensive, even if it scales favorably. For instance, when inserting a square into a two-dimensional segment tree, $8 \cdot (1 + \log n)$ nodes are created in the tree. This is a lot even for small n but does not increase significantly for large n .

7.4 Discussion

To summarize this section, we confirmed the estimation quality of 3KL and KSG across all dependency types tested. Additionally, we compared the performance of DEMI, ADEMI and REFS both on synthetic and real data. As expected, DEMI consistently outperforms the SE. The evaluation of ADEMI depends on the context and application. While it is slow for small window sizes, it barely slows down for larger sizes. On the one hand, this means that it is often recommendable to use DEMI if the data size is small. On the other hand, ADEMI can be used for very data-intensive

tasks such as monitoring high-throughput streams. A problem with stream monitoring often is the multiplicative cost of high temporal resolutions: A stream with frequent items permits less time to process a new item, and a window with fixed time length contains more items. This leads to increased time to process a new item. As we have seen, the second factor is nearly negligible when using ADEMI.

8 CONCLUSIONS

In this work we have studied the efficiency of estimating mutual information using nearest-neighbor distances. We have considered the estimator by Kraskov et al. [17](KSG) and the direct application of the entropy estimator [9, 16](3KL). We have investigated the computational complexity of these estimators on static data and have proven a tight lower bound for both in the algebraic computation tree model. Next, we have turned to the maintenance of 3KL and KSG estimates on dynamic data and have examined possible optimizations and limitations. We have inferred a lower bound for the computational complexity of this task. We also have presented two dynamic data structures DEMI and ADEMI that maintain 3KL and KSG estimates. We have proven that both data structures require asymptotically less time to update their estimate than the lower bound to recompute it. Additionally, for maintenance of 3KL estimates, the time complexity of ADEMI is near optimal. Finally, we have validated the performance of our approach empirically. We have shown that the 3KL has a good rate of convergence for various dependencies. We also have benchmarked our data structure using both synthetic and real data and have shown that ADEMI is very fast for large data sets.

Future Work. In this work we have focused on exact computations of nearest-neighbor based MI estimators for dynamic data. It remains open whether our approach offers the best trade-off between estimation quality and computation time.

For one, it would be interesting which results one could achieve by binning the data and using estimators for discrete distributions [5, 10]. However, it is unclear how the bin width should be chosen, given the evolving nature of a stream. If the bin width needs adjustment, this is computationally expensive or reduces the quality of estimation if there is no adjustment.

It would also be interesting to study which provable quality one might achieve with approximations. While there exist approximations of static estimators [3, 12], there are no bounds for additive or multiplicative errors. But these would be very important because there exists a lot of work comparing static estimators. In addition, empirical assessments of new estimators often cover only some of the dependencies that MI quantifies.

A FORMAL PROOFS

A.1 Proof of Theorem 4.1

THEOREM 4.1. *The problem 3KL-ESTIMATION has time complexity $\Omega(n \log n)$.*

PROOF. The proof is by reduction from the problem INTEGERELEMENTDISTINCTNESS. Given a multiset $A = \{a_1, \dots, a_n\}$ of integers, are there two indices $i \neq j$ such that $a_i = a_j$ are duplicates. The problem INTEGERELEMENTDISTINCTNESS has a known lower bound of $\Omega(n \log n)$ in the algebraic computation tree model [19]. For an instance A of INTEGERELEMENTDISTINCTNESS, we construct an instance of 3KL-ESTIMATION P as follows. For $a_i \in A$, the set P contains two points $p_i = (i, a_i + \frac{1}{4+i})$ and

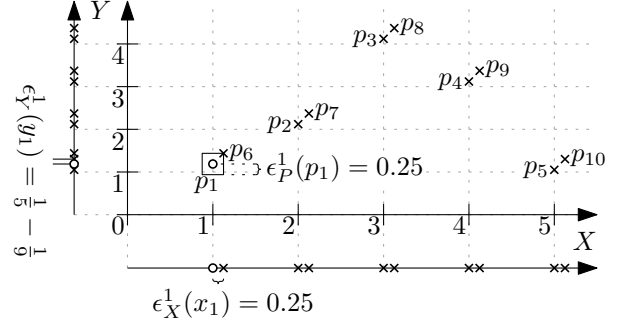


Figure 10: An illustration of the construction of P in Appendix A.1.

$p_{n+i} = p_i + (0.25, 0.25)$. Note that p_i and p_{n+i} are closer than any other pair, because i and a_i are integers. Additionally, we add the offset $\frac{1}{4+i}$ to the y -coordinates, because duplicates in A would otherwise lead to a nearest-neighbor distance of 0 and thus $\log(0)$ in Equation 4. Figure 10 features an example for this construction for the INTEGERELEMENTDISTINCTNESS instance $A = \{1, 2, 4, 3, 1\}$. The point p_1 is highlighted as circle and its nearest-neighbor distances are highlighted.

CLAIM 1. *A contains a duplicate if and only if $\widehat{I}_{3KL}(P) \neq \psi(|P|) - \psi(1)$ for $k = 1$.*

SUBPROOF. Let $i, j \in \{1, \dots, n\}$ be two integers with $i \neq j$. Based on the construction of P , it follows that $|x_i - x_j| = |x_{n+i} - x_{n+j}| \geq 1$. Additionally, it is $|x_i - x_{n+i}| = |y_i - y_{n+i}| = 0.25$. Using the reverse triangle inequality, it is $|x_i - x_{n+j}| \geq |x_i - x_j| - |x_j - x_{n+j}| \geq 0.75$. This holds for any $i \neq j$, which means that p_i is the nearest neighbor of p_{n+i} and vice versa, because we use the L_∞ norm. As a consequence the nearest neighbor distances are $\epsilon_P^1(p) = 0.25$ for all $p \in P$ and $\epsilon_X^1(x) = 0.25$ for all $x \in X$. Note that this means that the nearest neighbor distances in P and X are independent of the existence of duplicates in A .

If A does not contain any duplicates, it follows that $|y_i - y_j| = |y_{n+i} - y_{n+j}| \geq \frac{4}{5}$, since A only contains integers and the difference between $\frac{1}{4+i}$ and $\frac{1}{4+j}$ is less than 0.2. By the same arguments as above it follows that $|y_i - y_{n+j}| \geq 0.55$ and that $\epsilon_Y^1(y) = 0.25$ for all $y \in Y$. We can then use these values in Equation 4, which yields:

$$\widehat{I}_{3KL}(P) = \psi(|P|) - \psi(1) + \frac{1}{|P|} \sum_{m=1}^{|P|} \log \left(\frac{0.25 \cdot 0.25}{(0.25)^2} \right) = \psi(|P|) - \psi(1). \quad (8)$$

Conversely, if A contains the duplicates $a_i = a_j$, it is $|y_i - y_j| = |\frac{1}{4+i} - \frac{1}{4+j}| \leq 0.2$ and $|y_i - y_{n+j}| \geq |0.25 - \frac{1}{4+j}|$ and thus $\epsilon_Y^1(y_i) \leq 0.2$. Additionally, because of $j \neq i$ it also is $\epsilon_Y^1(y_i) > 0$. It follows that

$$\log \left(\frac{\epsilon_Y^1(y_i) \cdot \epsilon_X^1(x_i)}{(\epsilon_P^1(p_i))^2} \right) < 0 \Rightarrow \frac{1}{|P|} \sum_{m=1}^{|P|} \log \left(\frac{\epsilon_X^1(x_m) \cdot \epsilon_Y^1(y_m)}{\epsilon_P^1(p_m)^2} \right) < 0 \quad (9)$$

and analogously to Equation 8 we obtain $\widehat{I}_{3KL}(P) < \psi(|P|) - \psi(1)$. This concludes the subproof. ■

It is clear that P can be constructed in time $O(|A|)$, which means $|P| \in O(|A|)$. After computing $\widehat{I}_{3KL}(P)$, the result is only compared to a sum over $|P|$ numbers, because $\psi(|P|) - \psi(1) = \sum_{m=1}^{|P|-1} \frac{1}{m}$ by definition of the digamma function. Note that this

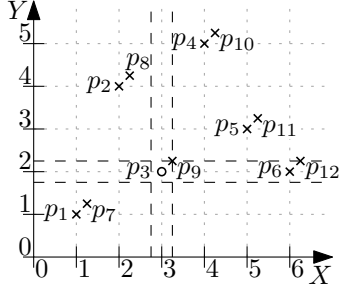


Figure 11: An illustration of the construction of P in Appendix A.2.

reduction works analogously for any fixed $k > 0$ by placing $k - 1$ points evenly spaced on the diagonal between each pair p_i and p_{n+i} . Because k is fixed, the size of P increases only by a constant factor. Therefore, the complexity of the reduction is in $O(n)$. This means that determining $\widehat{I_{3KL}}(P)$ has a lower bound of $\Omega(n \log n)$. \square

A.2 Proof of Theorem 4.3

THEOREM 4.3. *The problem KSG-ESTIMATION has a time complexity in $\Omega(n \log n)$.*

PROOF. Similarly to the Proof of Theorem 4.1, see Appendix A.1, we reduce the problem to INTEGERELEMENTDISTINCTNESS. For any instance A of INTEGERELEMENTDISTINCTNESS, we construct an instance of KSG-ESTIMATION P as follows. For $a_i \in A$, the set P contains two points $p_i = (i, a_i)$ and $p_{n+i} = (i + 0.25, a_i + 0.25)$. We use 0.25 because it means that this pair of points is closer than any other pair, because i and a_i are integers. Figure 11 features an example for this construction for the INTEGERELEMENTDISTINCTNESS instance $A = \{1, 4, 2, 5, 3, 2\}$. The dashed lines in the figure illustrate the areas of the marginal counts $C_x^1(p_3)$ and $C_y^1(p_3)$.

CLAIM 2. *A contains a duplicate if and only if $\widehat{I_{KSG}}(P) \neq \sum_{m=1}^{|P|-1} \left(\frac{1}{m}\right) - 1$ for $k = 1$.*

SUBPROOF. Let $i, j \in \{1, \dots, n\}$ be two integers with $i \neq j$. Based on the construction of P , it follows that $|x_i - x_j| = |x_{n+i} - x_{n+j}| \geq 1$. Additionally, it is $|x_i - x_{n+i}| = |y_i - y_{n+i}| = 0.25$. Using the reverse triangle inequality, it is $|x_i - x_{n+j}| \geq |x_i - x_j| - |x_j - x_{n+j}| \geq 0.75$. This holds for any $i \neq j$, which means that p_i is the nearest neighbor of p_{n+i} and vice versa, because we use the L_∞ norm. As a consequence, the marginal counts $C_x^1(p)$ are 1 for all $p \in P$, independent of the existence of duplicates in A .

If A does not contain any duplicates, it follows that $|y_i - y_j| = |y_{n+i} - y_{n+j}| \geq 1$, since A only contains integers. By the same arguments as above it follows that $|y_i - y_{n+j}| \geq 0.75$ and that $C_y^1(p) = 1$ for all $p \in P$. We can then use these values in Equation 6 and because of $\psi(x) = \sum_{m=1}^{x-1} \left(\frac{1}{m}\right) - C$ it is:

$$\widehat{I_{KSG}}(P) = \psi(1) + \psi(|P|) - \frac{1}{1} - \frac{1}{|P|} \sum_{m=1}^{|P|} \psi(1) + \psi(1) = \sum_{m=1}^{|P|-1} \left(\frac{1}{m}\right) - 1 \quad (10)$$

Conversely, if A contains the duplicates $a_i = a_j$, it is $|y_i - y_j| = 0$ and $|y_i - y_{n+j}| = 0.25$ and thus $C_y^1(p_i) \geq 3$. Because of $\psi(x+1) = \psi(x) + \frac{1}{x} > \psi(x)$ for all $x \geq 0$, it is, analogously to Equation 10, $\widehat{I_{KSG}}(P) < \sum_{m=1}^{|P|-1} \left(\frac{1}{m}\right) - 1$. This concludes the subproof. \blacksquare

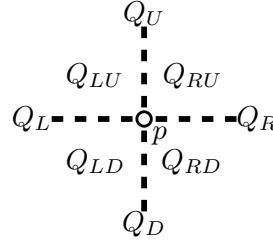


Figure 12: The partitioning of Q in Appendix A.3.

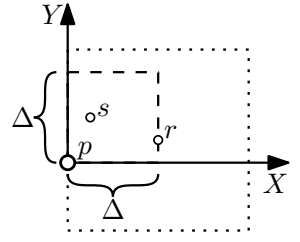


Figure 13: An example for Q_{RU} .

It is clear that P can be constructed in time $O(|A|)$, which means $|P| \in O(|A|)$. Note that this reduction works analogously for any fixed $k > 0$ by placing $k - 1$ points evenly spaced on the diagonal between each pair p_i and p_{n+i} . Because k is fixed, the size of P increases only by a constant factor. After computing $\widehat{I_{KSG}}(P)$, the result is only compared to a sum over $|P|$ numbers. Therefore the complexity of the reduction is in $O(n)$. This means that determining $\widehat{I_{KSG}}(P)$ has a lower bound of $\Omega(n \log n)$. \square

A.3 Proof of Theorem 5.1

THEOREM 5.1. *Let $P \subseteq \mathbb{R}^2$ be a set of points. For any point $p \in P$ there exist at most $8k$ points $q \in P$ such that p is one of the k nearest neighbors of q using the L_∞ -norm.*

PROOF. Let $p = (x, y) \in P$ be a point and $Q \subseteq P$ be the set of points such that for each point $q \in Q$, p is one of the k nearest neighbors of q . We separate Q into eight sets based on their relative location to p , as illustrated in Figure 12. There are four axis-aligned rays $Q_L, Q_R, Q_U, Q_D \subseteq Q$ centered at p such that points on any of these rays share one component with p and differ in the other one. Additionally, there are four quadrants $Q_{RU}, Q_{LU}, Q_{LD}, Q_{RD} \subseteq Q$ centered at p excluding the axis-aligned rays. Because p cannot be its own nearest neighbor, these eight sets partition Q . To prove the lemma we proceed to show that each of these eight sets contains at most k points.

Let $r = (x_r, y_r)$ be the most distant point to p in the axis-aligned ray Q_R , that is, $|x - x_r| = \max_{(x_i, y_i) \in Q_R} |x - x_i|$. Then all other points in Q_R are on the line between p and r and thus closer to r than p . This means that Q_R cannot contain more than k points, because p would not be a nearest neighbor of r otherwise. By symmetry, this result also holds for the sets Q_L, Q_U, Q_D .

Similarly, Let $r = (x_r, y_r)$ be the most distant point to p in the quadrant Q_{RU} and let Δ be that distance. More formally, it is

$$\Delta = \max(|x - x_r|, |y - y_r|) = \max_{(x_i, y_i) \in Q_{RU}} \max(|x - x_i|, |y - y_i|).$$

An exemplary illustration can be found in Figure 13 with the set $Q_{RU} = \{s, r\}$. For any other point $q_i = (x_i, y_i) \in Q_{RU}$ with $q_i \neq r$ it is $x < x_i \leq x + \Delta$ and $y < y_i \leq y + \Delta$, because r is the point most distant to p . Figure 13 illustrates this by delimiting the area in which all points of Q_{RU} lie with dashed lines. Because of $x_r > x$ and $y_r > y$, it follows that $|x_r - x_i| < \Delta$ and $|y_r - y_i| < \Delta$. This means that q_i is a nearest neighbor of r . Figure 13 shows this by highlighting the area of nearest neighbors of r with dotted lines. Analogously to the axis-aligned rays, Q_{RU} cannot contain more than k points, because p would not be a nearest neighbor of r otherwise. By symmetry, this result also holds for the sets Q_{LU}, Q_{LD}, Q_{RD} . \square

B ADEMI

Data Structure 5: ADEMI

```

struct {
  real  $x, y$ 
  real  $\epsilon_P^k, \epsilon_X^k, \epsilon_Y^k$ 
} DemiPoint;

struct {
  DemiPoint[ ]  $P_D$ 
  BST<DemiPoint*>  $T_x, T_y$ 
  real base, sum
  2D dynamic range tree  $T_{range}$ 
  2D dynamic segment tree  $T_{seg}$ 
} state;

```

Algorithm 6: ADEMI-INSERT(S_p, p_{n+1})

```

1 Compute  $\epsilon_P^k(p_{n+1})$   $O(k \cdot \log n \log \log n)$ 
2 Compute  $\epsilon_X^k(x_{n+1})$  and  $\epsilon_Y^k(y_{n+1})$   $O(\log n)$ 
3 Insert  $p_{n+1}$  into  $P_D$   $O(1)$ 
4 Reference  $p_{n+1}$  in  $T_x, T_y$   $O(\log n)$ 
5 Insert  $p_{n+1}$  into  $T_{range}$   $O(\log n \log \log n)$ 
6 Insert square( $p_{n+1}, P'$ ) into  $T_{seg}$   $O(\log n \log \log n)$ 
7  $base \leftarrow base + \frac{1}{n}$   $O(1)$ 
8  $sum \leftarrow sum + \log \left( \frac{\epsilon_X^k(x_{n+1}) \cdot \epsilon_Y^k(y_{n+1})}{(\epsilon_P^k(p_{n+1}))^2} \right)$   $O(1)$ 
9  $A \leftarrow \{p_i \in P : \max(|x_i - x_{n+1}|, |y_i - y_{n+1}|) < \epsilon_P^k(p_i)\}$   $O(\log n \log \log n + |A|)$ 
10  $B \leftarrow \{p_i \in P : |x_i - x_{n+1}| < \epsilon_X^k(x_i)\}$   $O(k \cdot \log n)$ 
11  $C \leftarrow \{p_i \in P : |y_i - y_{n+1}| < \epsilon_Y^k(y_i)\}$   $O(k \cdot \log n)$ 
12 forall  $p_i \in A$  do
13   Delete square( $p_i, P$ ) from  $T_{seg}$   $O(|A| \cdot \log n \log \log n)$ 
14   Compute  $\epsilon_P^k(p_i)$   $O(|A| \cdot k \cdot \log n \log \log n)$ 
15   Insert square( $p_i, P'$ ) into  $T_{seg}$   $O(|A| \cdot \log n \log \log n)$ 
16    $sum \leftarrow sum + \log((\epsilon_P^k(p_i))^2) - \log((\epsilon_P^k(p_i))^2)$   $O(|A|)$ 
17 forall  $p_i \in B$  do
18   Compute  $\epsilon_X^k(x_i)$   $O(|B| \cdot k \cdot \log n)$ 
19    $sum \leftarrow sum - \log(\epsilon_X^k(x_i)) + \log(\epsilon_X^k(x_i))$   $O(|B|)$ 
20 forall  $p_i \in C$  do
21   Compute  $\epsilon_Y^k(y_i)$   $O(|C| \cdot k \cdot \log n)$ 
22    $sum \leftarrow sum - \log(\epsilon_Y^k(y_i)) + \log(\epsilon_Y^k(y_i))$   $O(|C|)$ 

```

ACKNOWLEDGMENTS

This work was partially supported by the DFG Research Training Group 2153: "Energy Status Data – Informatics Methods for its Collection, Analysis and Exploitation"

REFERENCES

- [1] Michael Ben-Or. 1983. Lower bounds for algebraic computation trees. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing*, 80–86.
- [2] Jon Louis Bentley. 1977. *Algorithms for Klee's rectangle problems*. Technical Report. Technical Report, Computer.
- [3] Jonathan Boidol and Andreas Hapfelmeyer. 2017. Fast mutual information computation for dependency-monitoring on data streams. In *Proceedings of the Symposium on Applied Computing*. ACM, 830–835.
- [4] Brian Bole, Chetan S Kulkarni, and Matthew Daigle. 2014. Adaptation of an electrochemistry-based li-ion battery model to account for deterioration observed under randomized use. In *Proceedings of Annual Conference of the Prognostics and Health Management Society*, Vol. 29. <https://ti.arc.nasa.gov/c/25/>
- [5] Amit Chakrabarti, Graham Cormode, and Andrew McGregor. 2007. A near-optimal algorithm for computing the entropy of a stream. In *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '07)*. Society for Industrial and Applied Mathematics, 328–335.
- [6] Timothy M Chan. 2006. A dynamic data structure for 3-d convex hulls and 2-d nearest neighbor queries. In *Proceedings of the 17th annual ACM-SIAM Symposium on Discrete Algorithm (SODA'06)*. 1196–1202.
- [7] Kenneth Ward Church and Patrick Hanks. 1990. Word association norms, mutual information, and lexicography. *Computational Linguistics* 16, 1 (1990), 22–29.
- [8] Thomas M. Cover and Joy A. Thomas. 2006. *Elements of information theory* (2. ed. ed.). Wiley-Interscience, Hoboken, NJ.
- [9] Dafydd Evans. 2008. A computationally efficient estimator for mutual information. In *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, Vol. 464. The Royal Society, 1203–1215.
- [10] Nicholas JA Harvey, Jelani Nelson, and Krzysztof Onak. 2008. Sketching and streaming entropy via approximation theory. In *IEEE 49th Annual IEEE Symposium on Foundations of Computer Science (FOCS '08)*. IEEE, 489–498.
- [11] Sanjiv Kapoor and Michiel Smid. 1996. New techniques for exact and approximate dynamic closest-point problems. *SIAM J. Comput.* 25, 4 (1996), 775–796.
- [12] Fabian Keller, Emmanuel Müller, and Klemens Böhm. 2015. Estimating mutual information on data streams. In *Proceedings of the 27th International Conference on Scientific and Statistical Database Management (SSDBM'15)*. ACM.
- [13] Shiraj Khan, Sharba Bandyopadhyay, Auroop R. Ganguly, Sunil Saigal, David J. Erickson, Vladimir Protopopescu, and George Ostroouchov. 2007. Relative performance of mutual information estimation methods for quantifying the dependence among short and noisy data. *Phys. Rev. E* 76 (2007), 15. Issue 2.
- [14] Justin B Kinney and Gurinder S Atwal. 2014. Equitability, mutual information, and the maximal information coefficient. *Proceedings of the National Academy of Sciences* 111, 9 (2014), 3354–3359.
- [15] Yuliya Kopylova, Duncan A Buell, Chin-Tser Huang, and Jeff Janies. 2008. Mutual information applied to anomaly detection. *Journal of Communications and Networks* 10, 1 (2008), 89–97.
- [16] LF Kozachenko and Nikolai N Leonenko. 1987. Sample estimate of the entropy of a random vector. *Problemy Peredachi Informatsii* 23, 2 (1987), 9–16.
- [17] Alexander Kraskov, Harald Stögbauer, and Peter Grassberger. 2004. Estimating mutual information. *Phys. Rev. E* 69 (2004), 16. Issue 6.
- [18] M. Lichman. 2013. UCI Machine Learning Repository. (2013). <https://archive.ics.uci.edu/ml/machine-learning-databases/00235/>
- [19] Anna Lubiw and András Rácz. 1991. A lower bound for the integer element distinctness problem. *Information and Computation* 94, 1 (1991), 83–92.
- [20] Kurt Mehlhorn and Stefan Näher. 1990. Dynamic fractional cascading. *Algorithmica* 5, 1 (1990), 215–241.
- [21] Angeliki Papana and Dimitris Kugiumtzis. 2009. Evaluation of mutual information estimators for time series. *International Journal of Bifurcation and Chaos* 19, 12 (2009), 4197–4215.
- [22] Hanchuan Peng, Fuhui Long, and Chris Ding. 2005. Feature selection based on mutual information criteria of max-dependency, max-relevance, and min-redundancy. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 27, 8 (2005), 1226–1238.
- [23] Josien PW Pluim, JB Antoine Maintz, and Max A Viergever. 2003. Mutual-information-based registration of medical images: a survey. *IEEE Transactions on Medical Imaging* 22, 8 (2003), 986–1004.
- [24] Peng Qiu, Andrew J Gentles, and Sylvia K Plevritis. 2010. Reducing the computational complexity of information theoretic approaches for reconstructing gene regulatory networks. *Journal of Computational Biology* 17, 2 (2010), 169–176.
- [25] Thomas Schreiber. 1995. Efficient neighbor searching in nonlinear time series analysis. *International Journal of Bifurcation and Chaos* 5, 02 (1995), 349–358.
- [26] Claude Elwood Shannon. 1948. A mathematical theory of communication. *The Bell System Technical Journal* 27 (1948), 379–423, 623–656.
- [27] Kanat Tangwongsan, Martin Hirzel, Scott Schneider, and Kun-Lung Wu. 2015. General incremental sliding-window aggregation. *Proceedings of the VLDB Endowment* 8, 7 (2015), 702–713.
- [28] Martin Vejmelka and Kateřina Hlaváčková-Schindler. 2007. Mutual information estimation in higher dimensions: A speed-up of a k-nearest neighbor based estimator. In *International Conference on Adaptive and Natural Computing Algorithms (ICANNGA'07)*. 790–797.
- [29] Janett Walters-Williams and Yan Li. 2009. Estimation of mutual information: A survey. In *International Conference on Rough Sets and Knowledge Technology (RSKT'08)*. 389–396.
- [30] Dan E Willard. 1985. New data structures for orthogonal range queries. *SIAM J. Comput.* 14, 1 (1985), 232–253.

Finding All Maximal Connected s -Cliques in Social Networks

Rachel Behar

The Rachel and Selim Benin School of Computer Science and Engineering, Hebrew University of Jerusalem, Israel
 rachel.beharharr@mail.huji.ac.il

Sara Cohen

The Rachel and Selim Benin School of Computer Science and Engineering, Hebrew University of Jerusalem, Israel
 sara@cs.huji.ac.il

ABSTRACT

Cliques are commonly used for social network analysis tasks, as they are a good representation of close-knit groups of people. For this reason (as well as for others), the problem of enumerating, i.e., finding, all maximal cliques in a graph has received extensive treatment. However, considering only complete subgraphs is too restrictive in many real-life scenarios where “almost cliques” may be even more useful. Hence, the notion of an s -clique, a clique relaxation that allows every node to be at distance at most s from every other node, has been introduced. *Connected s -cliques* add the natural requirement of connectivity to the notion of an s -clique.

This paper presents efficient algorithms for finding all maximal connected s -cliques in a graph. We present a provably efficient algorithm, which runs in polynomial delay. In addition, we present several variants of the well-known Bron-Kerbosch algorithm for maximal clique generation. Extensive experimentation over both real and synthetic datasets shows the efficiency of our algorithms, and their scalability with respect to graph size, density, and choice of s .

1 INTRODUCTION

Maximal cliques have long been considered a key component in the analysis of social networks [34]. Cliques are indeed highly cohesive sets of nodes, and as such are used to detect close-knit overlapping communities [13, 29, 36]. For this reason (among others), there has been extensive work on algorithms for finding all maximal cliques in a given graph, e.g., [1, 6, 8, 11, 17].

While the notion of a clique captures a completely cohesive group of nodes within a graph, this definition is often overly restrictive. In practice, it is obvious that sets of nodes can represent cohesive groups even if several links are missing; for example, within a community not all pairs of people will be friends. In addition, as networks are often built from observation of empirical data, there may be real-life links that are missing within the data captured. Searching for groups of nodes that are cliques will miss highly related groups of nodes for which links have been omitted from the dataset. To overcome these limitations, relaxations to the notion of a clique have been studied [30].

One useful relaxation to the notion of a clique, called an s -clique, was introduced over 65 years ago [24] to describe and measure connectivity in social groups. Given a graph G , we say that a set of nodes U is an s -clique, where s is a (typically small) natural number, if every pair of nodes $u, v \in U$ is at distance at most s one from another in G . In particular, when $s = 1$, the notions of a clique and an s -clique coincide. An s -clique is *maximal* if it cannot be extended with additional nodes, while retaining the required distances property. Unlike cliques, s -cliques may be

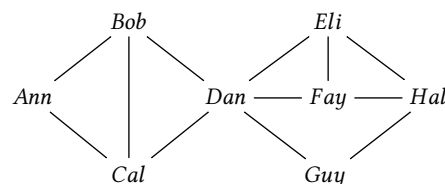


Figure 1: Example of a small social network G .

unconnected. Connected s -cliques add the natural requirement of connectivity.

Example 1.1. Consider the example of a small social network G in Figure 1. Graph G contains six maximal cliques, namely $\{a, b, c\}$, $\{b, c, d\}$, $\{d, e, f\}$, $\{e, f, h\}$, $\{d, g\}$, $\{g, h\}$, where a is a shorthand for Ann, b is a shorthand for Bob, and so on. This graph contains three maximal 2-cliques $\{a, b, c, d\}$, $\{b, c, d, e, f, g\}$ and $\{d, e, f, g, h\}$. Intuitively, the 2-cliques seem to better capture the graph communities, as they are a bit coarser. They also highlight the fact that d is a bridge between the communities.

Graph G contains two maximal 3-cliques $\{a, b, c, d, e, f, g\}$ and $\{b, c, d, e, f, g, h\}$, which (by their symmetric difference) indicate the people who, if linked, could help merge the communities. Thus, such a link might be suggested to Ann and Hal. Finally, we note that there is a single maximal 4-clique in G , as the diameter of G is four.

This paper studies the problem of enumerating (i.e., finding) all maximal connected s -cliques in a graph. Maximal clique enumeration has been shown to be useful in other areas (beyond social network analysis), e.g., finding subgraphs common to a set of input graphs [19], genome mapping and protein clustering in bioinformatics [14, 25], clustering for wireless sensor networks [4], and statistical analysis of financial networks [5]. As s -cliques are relaxations of cliques, they allow significantly greater flexibility, and may be useful for the above applications, e.g., to find subgraphs that are “almost common” to input graphs (i.e., that appear in slight variations in the various input graphs), to integrate genome mappings based on very similar subportions or to cluster protein sequences while allowing more flexibly for missing information.

An algorithm for enumerating maximal connected s -cliques can be used for new and interesting applications, such as link prediction in social networks [22], since missing direct links in large s -cliques are prime candidates for link suggestion. Note that large cliques could not be used for this purpose, as they are missing no links at all, by definition. Similarly, another possible application would be to help identify hidden connections in a social network, by finding maximal s -cliques that may form unidentified communities. We leave the development of such applications to future work, and focus in this paper on algorithms for efficiently enumerating all maximal s -cliques from a given graph.

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

The main contributions of this paper are three new algorithms for enumerating all maximal connected s -cliques from a given graph G . While it may seem from Example 1.1 that graphs have a small (polynomial) number of maximal s -cliques, this is in fact not always the case. As we demonstrate later, a graph may have exponentially many maximal s -cliques. (This was already well known for $s = 1$, and is true for larger values of s as well.) Hence, we cannot hope to derive a polynomial time algorithm for the problem at hand, as it may take exponential time to simply print the output. Instead, our first algorithm guarantees polynomial delay between results, i.e., the time to produce the first result, between every pair of subsequent results, and from the final result until completion, is polynomial.

The other two algorithms we present are adaptations of the well-known Bron-Kerbosch method, originally developed for finding all maximal cliques, to the problem at hand. While Bron-Kerbosch clique enumeration does not run in polynomial delay, it is known to be the fastest method, in practice, for maximal clique enumeration (when used with some specific optimizations). Hence, adapting this method to s -cliques is of interest. Optimizations for our adaptations, including pivoting and checking for feasibility, are studied. Extensive experimentation, over both real and synthetic datasets, proves the efficiency of our techniques, as well as their suitability for use over social network data.

2 RELATED WORK

Due to their numerous uses, the problem of finding all maximal cliques of a graph has received extensive attention. In the worst case, there can be exponentially many maximal cliques in a graph. In fact, [26] shows that the maximal number of cliques in graph with n nodes is $O(3^{n/3})$. Thus, the focus is on finding maximal cliques in time that is efficient with respect to the input and output. One well-known algorithm is that of Bron-Kerbosch [6], which, with the *pivoting* improvement of [32], guarantees a worst-time complexity of $O(3^{n/3})$ for graphs of size n . Hence the total time spent is no worse than required to return all maximal cliques on the graph with the most possible cliques.

Additional work on maximal clique enumeration has focused on output-efficient algorithms, i.e., algorithms whose runtime is a function of the number of maximal cliques in the given input graph [1, 17]. Several works have studied enumeration over sparse graphs [7, 12], as such graphs tend to be common in practice. Recent work has also focused on maximal clique enumeration over uncertain graphs [38], and over massive networks [9, 11].

Enumeration of maximal graphs for several relaxations of the notion of a clique has also been studied. The problem of mining all maximal k -plexes was studied in [3, 35], and enumeration of maximal c -isolated cliques was studied in [15]. There has also been work on mining quasi-cliques (i.e., densest subgraphs) in a single graph [23, 33, 37], and over a set of graphs [16], as well as mining locally dense subgraphs [31].

Among clique relaxations, both quasi-cliques and s -clubs appear to be most related to s -cliques. Formally, *quasi-cliques* are parameterized by a value γ , i.e., a subset S of nodes in a graph G is a γ -quasi-clique if every node in S is connected to at least $\gamma(|S| - 1)$ nodes in S . It has been shown [16] that there is a strong relationship between the parameter γ , and the diameter of the induced subgraph of G on S . For example, if $\frac{1}{2} \leq \gamma \leq \frac{|S|-2}{|S|-1}$, then the induced subgraph on S will have diameter at most 2. At first glance, it would seem then that this property can be utilized to

enumerate (connected) s -cliques, e.g., by enumerating γ -quasi-cliques with an appropriate (s -dependent) choice of γ . In fact, this is not the case, and previous algorithms for γ -quasi-cliques do not enumerate s -cliques. The difference is subtle, as every pair of nodes in an s -club S is of distance at most s in G , but may be of larger distance in the graph induced by S .

A subset U of nodes in a graph G is an s -club, if the diameter of U is at most s , i.e., if there is a path between every two nodes in U that only traverses nodes in U , that is of length at most s . This definition is different from that of s -cliques, where the distance between nodes is determined by the shortest path in the entire graph G . Similar to the maximum clique problem, the problem of finding an s -club of maximum size is also NP-complete [2]. However, unlike cliques and s -cliques, s -clubs are not hereditary (i.e., a subgraph of an s -club is not necessarily an s -club), and indeed s -club maximality testing is NP-complete [28]. Since maximal s -clubs cannot be efficiently recognized, enumerating maximal s -clubs cannot be achieved in polynomial delay (unlike enumerating maximal connected s -cliques, as we show in this paper).

The enumeration problem for connected s -cliques has not yet been studied. However, the optimization problem for s -cliques, i.e., the NP-complete problem of finding an s -club of maximal size, was studied in [2]. Enumerating s -cliques with given labels, over a labeled graph, is also NP-complete problem, and is studied in [18]. Finally, [28] has shown that for graphs with some special properties, the notions of connected s -cliques and s -clubs coincide.

3 FORMAL FRAMEWORK

Graphs and Induced Subgraphs. We use G, H (possibly with subscripts or superscripts) to denote simple undirected graphs. We use $V(G)$ to denote the nodes of G and $E(G)$ to denote the edges of G . Note that an edge is a pair $\{v, u\}$ where v and u are two different nodes in $V(G)$.

We will often be interested in *induced subgraphs* of a given graph G . Formally, a subset of nodes $U \subseteq V(G)$ defines the *induced subgraph* $G[U]$ of G consisting of precisely the set of nodes U , and the edges in $E(G)$ that are incident only on nodes in U . In notation, we have $V(G[U]) = U$ and $E(G[U]) = E(G) \cap U^2$. We say that H is an induced subgraph of G if $H = G[U]$, for some U , and denote this fact by $H \sqsubseteq G$.

We use $dist_G(u, v)$ to denote the number of edges on the shortest path between u and v in G and $N_G^i(v)$ to denote the set

$$N_G^i(v) := \{u \mid dist_G(v, u) \leq i \text{ and } u \neq v\}$$

of the nodes at distance at most i from v in G . We use $N_G(v)$ for the special case where $i = 1$, i.e., $N_G(v)$ contains all direct neighbors of v . Extending this notation, we use $N_G^{v,i}(V)$ and $N_G^{\exists,i}(V)$ to denote the set of nodes at distance at most i from all and at least one, respectively, v in V , i.e.,

$$N_G^{v,i}(V) := \{u \mid \forall v \in V, dist_G(v, u) \leq i \text{ and } u \notin V\},$$

$$N_G^{\exists,i}(V) := \{u \mid \exists v \in V, dist_G(v, u) \leq i \text{ and } u \notin V\}.$$

If G is clear from the context, we will omit the subscript.

Example 3.1. Consider graph G from Figure 1. Let $V = \{e, h\}$. We have:

$$\begin{aligned} N^{\exists,1}(V) &= \{d, f, g\} & N^{v,1}(V) &= \{f\} \\ N^{\exists,2}(V) &= N^{\exists,1}(V) \cup \{b, c\} & N^{v,2}(V) &= N^{\exists,1}(V). \end{aligned}$$

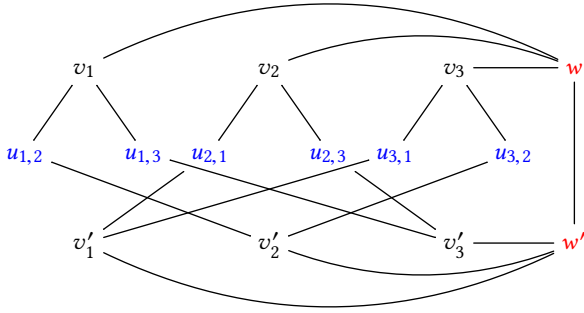


Figure 2: Graph G' .

Cliques and s -Cliques. A set of nodes U in a graph G is a *clique* if all pairs $u, v \in U$ are adjacent one to another in G . Cliques are useful in many problem areas, as they represent fully cohesive portions of G . As discussed earlier, in many scenarios, the requirement that a set of nodes form a clique may be overly restrictive.

Previous work [30] has studied various relaxations of the notion of a clique. In the following, let s be an integer and U be a set of nodes in a graph G . We say that

- U is an s -clique if, for all $u, v \in U$, it holds that $\text{dist}_G(u, v) \leq s$.
- U is a *connected s -clique* if U is an s -clique and the induced graph $G[U]$ is connected.

When $s = 1$, cliques coincide with both s -cliques and connected s -cliques.

Example 3.2. Consider graph G from Figure 1. The set of nodes $\{a, b, c, d, e, f, g\}$ is a (connected) 3-clique, but is not a 2-clique, e.g., $\text{dist}_G(a, f) = 3 > 2$. The set of nodes $\{a, b, c, d\}$ is a connected 2-clique, and the set $\{a, d\}$ is a 2-clique, but is not a connected 2-clique.

We say that U is a *maximal (connected) s -clique* in G , if U is a (connected) s -clique, and for all U' such that $U \subseteq U'$, it holds that U' is not a (connected) s -clique. It is natural to focus on *maximal (connected) s -cliques*. Indeed, every (connected) s -clique is contained in some maximal (connected) s -clique. Hence, maximal (connected) s -cliques can be viewed as a succinct representation for all (connected) s -cliques.

Example 3.3. Consider graph G' , from Figure 2. Let

$$\begin{aligned} V &= \{v_1, v_2, v_3\} & U &= \{u_{1,2}, u_{1,3}, u_{2,1}, u_{2,3}, u_{3,1}, u_{3,2}\} \\ V' &= \{v'_1, v'_2, v'_3\} & W &= \{w, w'\} \end{aligned}$$

Now, it is easy to observe that every subset C of $V \cup V'$ that does not contain both v_i and v'_i for some $i \leq 3$ is a 2-clique. Such 2-cliques are not maximal, however. A subset $C \subseteq V \cup V' \cup W$ will be a maximal connected 2-clique if (1) C contains precisely one among v_i, v'_i for each $i \leq 3$, and (2) C contains w, w' . Thus, for example, $\{v_1, v_2, v'_3\}$ is a 2-clique (but not maximal nor connected), and $\{v_1, v_2, v'_3, w, w'\}$ is a maximal connected 2-clique in G . Note that there are additional ways to form maximal connected 2-cliques, when taking nodes from U . For example, $\{v_1, v'_2, w, w', u_{1,2}\}$ is also a maximal connected 2-clique.

We can now formally state our problem of interest: *Given a graph G and an integer s , enumerate (i.e., find, one after another) all maximal connected s -cliques in G .* As larger s -cliques can, naturally, be more interesting, we will also briefly consider a

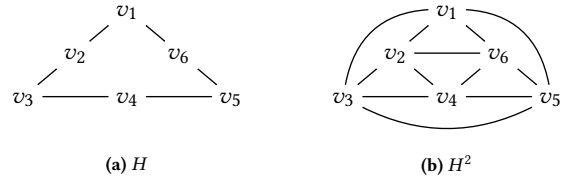


Figure 3: Graph H and corresponding graph H^2 .

related problem, i.e., that of finding maximal s -cliques of size at least k , for some given number k .

REMARK 1. We do not consider enumeration of maximal s -cliques that are not necessarily connected. This is because enumeration of maximal s -cliques over a graph G can be reduced to maximal clique enumeration: Define G^s as the graph containing an edge between nodes u, v if they are of distance at most s in G . Then, the maximal cliques in G^s are precisely the maximal s -cliques in G . However, this reduction is not applicable for connected s -cliques, as cliques in G^s can correspond to unconnected sets in G . Hence, enumeration of maximal connected s -cliques is more difficult, as the following example demonstrates.

Consider the graphs H, H^2 in Figures 3 (a) and (b). The graph H^2 contains an edge between every pair of nodes in H that are of distance at most 2 one from another. Every 2-clique in H is a clique (in the standard sense) in H^2 . Observe, for example, that the sets $C_1 = \{v_1, v_2, v_6\}$ and $C_2 = \{v_1, v_3, v_5\}$ are 2-cliques in H and cliques in H^2 . Unlike the set C_1 , the set C_2 is not a connected 2-clique. Indeed no two nodes in C_2 are connected in H , and thus, C_2 forms an unconnected subgraph of H . This cannot be seen when looking at H^2 alone; the information about connectedness is lost in the given graph transformation.

We note that due to the fact that the number of sets in the result can be exponential in the size of G , the problem of enumerating all maximal connected s -cliques cannot be solved in polynomial time. Hence, exponential time may be needed just to print the output. Therefore, we focus on finding algorithms whose runtime is either provably efficient with respect to the output size (e.g., polynomial delay) or of high efficiency in practice.

Example 3.4. We demonstrate a graph with exponentially many maximal connected s -cliques for $s = 2$. It is easy to extend this idea to derive a graph with exponentially many maximal connected s -cliques for other values of s . Let n be an integer. Let

$$\begin{aligned} V &= \{v_i \mid i \leq n\} & U &= \{u_{i,j} \mid i \neq j \leq n\} \\ V' &= \{v'_i \mid i \leq n\} & W &= \{w, w'\} \end{aligned}$$

be sets of nodes. We add edges $\{v_i, u_{i,j}\}, \{u_{i,j}, v'_j\}$ for all $i \neq j \leq n$, as well as edges $\{v_i, w\}, \{v'_i, w'\}$ for all $i \leq n$. Finally, we add the edge $\{w, w'\}$. Graph G' from Figure 2 has precisely this structure, for $n = 3$.

Every pair of nodes v_i, v'_j where $i \neq j$ have distance 2 one from another, while v_i, v'_i have distance 3. Nodes w, w' are at distance at most 2 from every node in the graph. Thus, it is easy to see that every choice of nodes including precisely one among v_i, v'_i , for all $i \leq n$, as well as nodes w, w' , yields a maximal connected 2-clique. (Note that nodes $u_{i,j}$ cannot be added to such sets.) Thus, the graph derived has at least 2^n maximal connected 2-cliques, while it has only $2n + n(n-1) + 2$ nodes, i.e., the number of maximal connected 2-cliques is exponential in the size of the graph.

Algorithm POLYDELAYENUM(G, s)

```

1.  $Q \leftarrow \text{EMPTYQUEUE}()$ 
2.  $I \leftarrow \text{EMPTYINDEX}()$ 
3.  $C \leftarrow \text{EXTENDMAX}(\emptyset, G, s)$ 
4.  $\text{ENQUEUE}(Q, C)$ 
5.  $\text{INSERT}(I, C)$ 
6. while NOTEMPTY( $Q$ )
7.   do  $C \leftarrow \text{DEQUEUE}(Q)$ 
8.      $\text{PRINT}(C)$ 
9.     for  $v \in N_G^{\exists,1}(C)$ 
10.       $C' \leftarrow \text{EXTENDMAX}(\{v\}, G[C \cup \{v\}], s)$ 
11.       $C'' \leftarrow \text{EXTENDMAX}(C', G, s)$ 
12.      if  $C'' \notin I$ 
13.        then  $\text{ENQUEUE}(Q, C'')$ 
14.         $\text{INSERT}(I, C'')$ 

```

Algorithm EXTENDMAX(C, G, s)

```

1. if  $C = \emptyset$ 
2.   then add an arbitrary node to  $C$ 
3.   while  $\exists v \in N_G^{v,s}(C) \cap N_G^{\exists,1}(C)$ 
4.     do  $C \leftarrow C \cup \{v\}$ 
5.   return  $C$ 

```

Figure 4: An polynomial delay algorithm for enumerating all maximal connected s -cliques.

4 A POLYNOMIAL DELAY ALGORITHM

We present a provably efficient algorithm for enumerating all maximal connected s -cliques in a given graph G . This algorithm is inspired by the general purpose algorithm for enumerating maximal subgraphs satisfying some connected-hereditary property, appearing in [10]. Our algorithm, called POLYDELAYENUM, appears in Figure 4.

The algorithm POLYDELAYENUM uses two data structures:

- Q , a queue, containing maximal connected s -cliques that must still be processed. Later, in Section 6, we will also consider using a priority queue for Q .
- I , an index containing maximal connected s -cliques that have already been generated. In order to achieve the required runtime, access to I (both insertions and membership checks) must be in time that is at most logarithmic in the size of I . Thus, for example, I can be implemented as a BTree.

POLYDELAYENUM uses a sub-procedure, called EXTENDMAX, which is given, as input, a set C , a graph G and the integer s . We note that the set C provided as input is always a connected s -clique. EXTENDMAX returns a set C' such that

- $C \subseteq C'$ (this is ensured as C' is created by adding nodes to C);
- C' is a connected s -clique (as we only add a node that is connected to and of distance at most s from the nodes presently in C);
- C' is maximal in G , with respect to the above two properties (as we continue to add nodes as long as possible).

For example, when calling EXTENDMAX with the empty set (Line 3 of POLYDELAYENUM), a single maximal connected s -clique is returned. In general, the output of EXTENDMAX may differ, depending on the order in which we iterate over the nodes of G . In the special case that G is almost a connected s -clique, i.e., contains a single node that contradicts this property (as occurs in the

invocation of EXTENDMAX in Line 10 of POLYDELAYENUM), there is only one possible output for EXTENDMAX.

Now, POLYDELAYENUM begins by finding a single maximal connected s -clique, using EXTENDMAX, and adding this set to both Q and I (Lines 1–5). While Q is not empty, we remove a maximal connected s -clique C from Q , and print C (Lines 6–8). Now, for each node v that is a neighbor of some node in C , we proceed as follows. First (Line 10) we find the s -clique C' containing v that is maximal with respect to $G[C \cup \{v\}]$. Next (Line 11), we extend this set so as to derive an s -clique C'' that is maximal with respect to G . If we have not created C'' yet (i.e., $C'' \notin I$), we add it to Q and I (Lines 12–14). We note that a key aspect of the algorithm is the fact that EXTENDMAX is called twice, consecutively (Lines 10 and 11) with different graphs as input. The first invocation guides the creation of a new connected s -clique C' to contain portions from C . The second invocation ensures maximality with respect to the input graph G .

Example 4.1. Consider calling POLYDELAYENUM with the graph G from Figure 1 and $s = 2$. At first, an arbitrary 2-clique will be created, such as $C = \{a, b, c, d\}$. This set C will be added into the queue Q . When removed from the queue (Line 7), we will print C and then iterate over the set $N_G^{\exists,1}(C) = \{e, f, g\}$. Consider the case where we choose $v = e$ (in Line 9). We will call EXTENDMAX($\{e\}, G[C \cup \{e\}], 2$), deriving the set $C' = \{b, c, d, e\}$. There is only one way to extend C' in order to derive a maximal connected 2-clique. This extension will be returned from the next call to EXTENDMAX in Line 11, $C'' = \{b, c, d, e, f, g\}$, and inserted into the queue. Execution will proceed similarly, first for all other neighbors of C , which will return the same result C'' , that will not be enqueued again. Then, the next graph dequeued will be C'' , that will form the last s -clique $\{d, e, f, g, h\}$ when h is added.

The following result holds. (In practice, the delay between answers is typically lower than the theoretical result.)

THEOREM 4.2. *Given a connected graph G and an integer s , the algorithm POLYDELAYENUM prints every maximal connected s -clique in G , precisely once. In addition, the delay before printing the first answer, and between every two consecutive answers, and from the time the last answer is printed until termination, is $O(|V(G)|^3)$.*

PROOF. We start by showing correctness of the algorithm. First, it is immediate from the structure of EXTENDMAX that every set printed must be a maximal connected s -clique. Second, observe that every set is printed at most once, as we only insert C into Q if it was not already generated in the past (i.e., does not appear in I). It remains to show that every maximal connected s -clique is indeed printed.

Let v be an arbitrary node in G . Then, there is some set C , printed by POLYDELAYENUM, that contains v . (This can be shown by induction on the distance of v from the closest node in the first set generated by POLYDELAYENUM.)

We can now prove that every maximal connected s -clique is printed. Let C_* be some maximal connected s -clique. Let C_m be the set printed by POLYDELAYENUM for which the largest connected component in $C_* \cap C_m$ is maximal. (If there are ties, choose C_m arbitrarily among all such sets.) If $C_m = C_*$, we have indeed printed C_* . Suppose otherwise. Observe first that $C_m \cap C_*$ cannot be empty, as POLYDELAYENUM must print some set containing each node in C_* . Now, since $C_m \neq C_*$, there is some node $v \in C_* - C_m$, such that v is connected to the largest connected component in $C_m \cap C_*$. After dequeuing C_m from Q in Line 7, we will iterate over the node v in Line 9. Then, Line 10 will return a

Algorithm CLIQUES(R, P, X)

1. **if** $P = \emptyset$ and $X = \emptyset$
2. **then** PRINT(R)
3. **for** $v \in P$
4. **do** CLIQUES($R \cup \{v\}, P \cap N(v), X \cap N(v)$)
5. $P \leftarrow P - \{v\}$
6. $X \leftarrow X \cup \{v\}$

Figure 5: Bron-Kerbosch algorithm for enumerating maximal cliques.

set C' containing v , as well as the largest connected component in $C_m \cap C_*$ (since these together form a connected s -clique). This set will be enlarged to C'' in Line 11. Now, C'' is either inserted into Q (and eventually printed) or was already inserted in Q (and eventually printed). Observe that the largest connected component in $C'' \cap C_*$ must be larger than that in $C_m \cap C_*$, which contradicts the maximality of choice of C_m . Hence, it must be $C_m = C_*$, and C_* is indeed printed.

We will now show that the delay is $O(|V(G)|^3)$. First, note that by a preprocessing step we can compute $N_G^s(v)$ for every node $v \in V(G)$ in time $O(|V(G)|^3)$. In addition, EXTENDMAX runs in $O(|V(G)|^2)$, since it traverses each edge at most once, in increasing distance from the set C . Now, the delay before printing the first answer is determined by (1) the preprocessing step which finds the distances between nodes and (2) running EXTENDMAX in Line 3. Therefore the delay before printing the first answer is $O(|V(G)|^3)$.

The delay between every two consecutive answers, and from the time the last answer is printed until termination, is determined by the runtime of a single iteration of the while loop in Line 6 in POLYDELAYENUM, because in each iteration a single answer is printed and the last answer is printed in the last iteration. In each iteration, the for loop in Line 9 goes over the set $N_G^{\leq 1}(C)$, that is of size $O(|V(G)|)$, and for each node v in the set calls EXTENDMAX, which runs in $O(|V(G)|^2)$. In total we derive that each iteration runs in $O(|V(G)|^3)$. \square

5 ADAPTATION OF BRON-KERBOSCH ALGORITHM

We start by reviewing the Bron-Kerbosch algorithm for enumerating maximal cliques. We then present strategies and optimization techniques to adapt this algorithm for enumerating maximal connected s -cliques.

5.1 Maximal Cliques

The Bron-Kerbosch algorithm for enumerating maximal cliques appears in Figure 5. This algorithm, called CLIQUES, recursively searches for all maximal cliques. When called with sets R, X and P , it searches for all maximal cliques containing all nodes in R , possibly some nodes in P , and no nodes in X . In particular, in the first invocation, we send the empty set for R and X , and the set $V(G)$ for P .

Throughout the execution, R is always a clique. The sets P and X are disjoint, and always satisfy that $P \cup X$ contains precisely every node that is connected to all nodes in R (i.e., those can potentially be used to extend R). Therefore, when P and X are both empty, R is a maximal clique. Otherwise, if P is not empty, the algorithm attempts to add each node $v \in P$ in turn to R , updating P and X to contain only nodes that are connected to

v (in addition to being connected to the rest of R). After the recursive call that includes v , the node v is added to X , so as to create additional unseen cliques—those that exclude v . (Cliques containing v will be produced in the recursive call.)

The algorithm CLIQUES, as presented in Figure 5, runs quite poorly in practice. To improve the runtime, [32] presented a technique called *pivoting* to reduce the branching factor of the recursion, by iterating over only a subset of P . In particular, instead of iterating over P , their algorithm:

- first, chooses a *pivot* node $u \in P \cup X$.
- then, iterates only over the nodes v in $P - N(u)$.

The intuition behind this improvement is that, for any node u , every maximal clique must either contain u , or contain some node that is not a neighbor of u . (Otherwise, if the maximal clique contains only neighbors of u , it must also contain u .) Hence, it is sufficient to iterate over the nodes in P , other than the neighbors of u . The pivot node is chosen so as to minimize the set $P - N(u)$. This optimization ensures a worst case runtime of $O(3^{n/3})$, which is order of the largest number of cliques possible in a graph of n nodes. Previous work has shown that this improvement causes the algorithm to run very well in practice.¹

5.2 Maximal Connected s -Cliques

We adapt the Bron-Kerbosch algorithm to return all maximal connected s -cliques, instead of all maximal cliques. There are, perhaps, two different natural approaches to adapt the algorithm CLIQUE to this new setting:

- (1) **R is always a connected s -clique:** In the CLIQUE algorithm, we have seen that R is always a clique. It is natural to adapt this algorithm so as to preserve the invariant that R is always a connected s -clique. This approach is taken in CsCLIQUES1 in Figure 6. Thus, when choosing a node v with which to extend R (Line 3), we will only choose nodes from P that are adjacent to some node in R . After choosing v , we perform a recursive call, with v added to R and we intersect sets P and X with the nodes of distance at most s from v . Thus, throughout the algorithm, P and X always contain precisely the nodes at distance at most s from every node in R . Only such nodes may possibly be used to extend R in the future.

Note the final change in the algorithm, in the condition for printing R (Line 1). We print R only if it is maximal, i.e., there are no nodes at distance at most s from R (i.e., nodes in P or X) that are adjacent to R .

- (2) **R is always an s -clique, but may be unconnected:** In the second adaptation, we do not require R to be connected. In the algorithm CsCLIQUES2 in Figure 7, the set R may be unconnected. To observe this, see that in Line 3 we can add any node v from P to R , even if v is not adjacent to R . Thus, R will be an s -clique throughout the execution, but may be unconnected. This requires an additional change to the condition for printing R in Line 1—we print R only if it is connected.

Example 5.1. Consider using CsCLIQUES1 to find all maximal connected 2-cliques for graph H in Figure 3. When first called, $R = X = \emptyset$, $P = \{v_1, \dots, v_6\}$ and $s = 2$. Since, R is empty, we iterate over all nodes in P . Suppose the first node chosen is v_1 .

¹Additional optimizations to the algorithm have been considered in the past, such as iterating over v in the outermost recursion according to a degeneracy ordering of G [12] and early recognition of special branching cases [27]. Such optimizations can also be included in our algorithm.

Algorithm CsCLIQUES1(R, P, X, s)

1. **if** $P \cap N^{\exists,1}(R) = \emptyset$ and $X \cap N^{\exists,1}(R) = \emptyset$
2. **then** PRINT(R)
3. **for** $v \in P \cap N^{\exists,1}(R)$
 - ▷ If $R = \emptyset$, then take $N^{\exists,1}(R) = G(V)$
4. **do** CsCLIQUES1($R \cup \{v\}, P \cap N^s(v), X \cap N^s(v), s$)
5. $P \leftarrow P - \{v\}$
6. $X \leftarrow X \cup \{v\}$

Figure 6: Adaptation of Bron-Kerbosch algorithm for enumerating all maximal connected s -cliques. Throughout the execution, R is always a connected s -clique.

Algorithm CsCLIQUES2(R, P, X, s)

1. **if** $P \cap N^{\exists,1}(R) = \emptyset$ and $X \cap N^{\exists,1}(R) = \emptyset$ and R is connected
2. **then** PRINT(R)
3. **for** $v \in P$
4. **do** CsCLIQUES2($R \cup \{v\}, P \cap N^s(v), X \cap N^s(v), s$)
5. $P \leftarrow P - \{v\}$
6. $X \leftarrow X \cup \{v\}$

Figure 7: Adaptation of Bron-Kerbosch algorithm for enumerating all maximal connected s -cliques. Throughout the execution, R is always an s -clique, but may be unconnected.

In the recursive call to CsCLIQUES1, we will have $R = \{v_1\}$, $P = \{v_2, v_3, v_5, v_6\}$ and $X = \emptyset$. When a recursive call is made for the second node v_2 , we will have $R = \{v_2\}$, $P = \{v_3, v_4, v_6\}$ and $X = \{v_1\}$. (Note that v_1 was removed from P in Line 5 and added to X in Line 6, in the previous iteration.) Now, consider the execution of CsCLIQUES1($\{v_2\}, \{v_3, v_4, v_6\}, \{v_1\}, 2$). We will iterate over nodes that are neighbors of $\{v_2\}$ in $\{v_3, v_4, v_6\}$, i.e., only over v_3 . This will cause a single recursive call to CsCLIQUES1($\{v_2, v_3\}, \{v_4\}, \{v_1\}, 2$), which will eventually produce the maximal connected 2-clique $\{v_2, v_3, v_4\}$.

We contrast this execution with the execution of algorithm CsCLIQUES2. At first, the algorithms will proceed in the same fashion, as $R = \emptyset$ and both algorithms will iterate over all nodes in the graph. However, consider the execution of the recursive call to CsCLIQUES2($\{v_2\}, \{v_3, v_4, v_6\}, \{v_1\}, 2$). Instead of iterating only over neighbors of v_2 in $\{v_3, v_4, v_6\}$, we will iterate over all nodes in this set. Assuming the order of iteration is $v_3 < v_4 < v_6$, this will cause two additional recursive calls: CsCLIQUES2($\{v_2, v_4\}, \emptyset, \{v_3\}, 2$) and CsCLIQUES2($\{v_2, v_6\}, \emptyset, \{v_3, v_4\}, 2$). Neither of these calls will produce maximal connected s -cliques.

When comparing these two approaches, it is immediately obvious that CsCLIQUES2 does extra work that is avoided by CsCLIQUES1, as CsCLIQUES2 can create many sets that will never be printed, as they are unconnected. The algorithm CsCLIQUES1 completely avoids this by ensuring that R is connected throughout its execution. Notwithstanding the fact that CsCLIQUES1 would seem to be much superior to CsCLIQUES2, in fact, we will see that the latter is much more amenable to optimizations (including the pivoting technique). This will be discussed further in Section 5.3. However, before discussing this further, we prove correctness of both algorithms.

In order to prove correctness, let $<$ be an arbitrary ordering of the nodes in G . We assume that the iteration over (the subset of) P in Line 3 of both CsCLIQUES1 and CsCLIQUES2 follows this ordering, i.e., if $v, v' \in P$ (resp. $v, v' \in P \cap N^{\exists,1}(R)$) and $v < v'$, then we will choose to iterate over v before iterating over v' . The ordering $<$ implies two types of total orderings over nodes in a connected s -clique C . Let $\omega_2(C) = v_1, \dots, v_k$ be the total ordering over C , defined by $<$. Let $\omega_1(C) = u_1, \dots, u_k$ be the total ordering over C , defined as follows:

- $u_1 = v_1$;
- for all $i > 1$, it holds that $u_i = v_j$ where v_j in the first node according to $\omega_2(C)$ that is not already among u_1, \dots, u_{i-1} , for which $G[\{u_1, \dots, u_i\}]$ is connected.

Example 5.2. Consider graph G' from Figure 2. Let $<$ be the total ordering that orders the nodes as seen in the graph from left to right, top to bottom, i.e.,

$$v_1 < \dots < w < u_{1,2} < \dots < u_{3,2} < v'_1 < \dots < w'.$$

Consider the set $C = \{v_1, v'_2, w, w', u_{1,2}\}$. Then, we have $\omega_1(C) = v_1, w, u_{1,2}, v'_2, w'$.

We consider the *execution tree* \mathcal{T}_i formed by (recursive) calls to CsCLIQUES $_i$, for $i = 1, 2$. To be precise, this tree has nodes of the form (R, P, X) where R, P, X are subsets of $V(G)$. An edge from a node (R, P, X) to a node (R', P', X') is labeled by a node v . This tree is defined recursively as follows: The root of \mathcal{T}_i is $(\emptyset, V(G), \emptyset)$. A node (R, P, X) has a child (R', P', X') with a connecting edge labeled v if the call to CsCLIQUES $_i$ with parameters (R, P, X) results in a recursive call with parameters (R', P', X') when the node v is chosen in Line 3.

Given a series of nodes $\bar{u} = u_1, \dots, u_n$, we say that \bar{u} is a path in \mathcal{T}_i if there is a path of edges starting from the root of \mathcal{T}_i with labels u_1, \dots, u_n (precisely in that order). A node (R, P, X) in \mathcal{T}_i is an *output node* if R is printed by CsCLIQUES $_i$, during the call with parameters (R, P, X) .

Several important properties of \mathcal{T}_i hold.

LEMMA 5.3. *Let G be a graph and let \mathcal{T}_i be the execution tree derived by calling CsCLIQUES $_i(\emptyset, V(G), \emptyset, s)$. Let (R, P, X) be a node in \mathcal{T}_i .*

- (1) *The set R is an s -clique;*
- (2) *If $i = 1$, the set R is connected;*
- (3) *$P \cup X = N^{V,s}(R)$;*
- (4) *Let u_1, \dots, u_k be the path to (R, P, X) in execution tree \mathcal{T}_i . If u_1, \dots, u_k is a prefix of $\omega_i(C)$ for some maximal connected s -clique C , then $C - R \subseteq P$;*
- (5) *For every other node (R', P', X') in \mathcal{T}_i it holds that $R \neq R'$;*
- (6) *If C is a maximal connected s -clique, then $\omega_i(C)$ is a path in \mathcal{T}_i .*

PROOF. We show Properties 1–4 by induction on the depth (i.e., distance from the root) of the node (R, P, X) . For base case of distance 0, i.e., the root node, all four properties are immediate. (For Property 3, note that $R = \emptyset$, and thus, every node in G is of distance at most s from all nodes in R . Indeed $P = V(G)$).

Now, assume that Properties 1–4 hold for nodes of distance at most k from the root. We prove the required for nodes at distance $k + 1$. Let (R, P, X) be a node at distance $k + 1$ from the root, and let (R', P', X') be its parent. Let v be the label of the edge from (R', P', X') to (R, P, X) . By the induction hypothesis, R' is an s -clique, if $i = 1$, R' is connected, and every node in P' is of distance at most s from all nodes in R' . Since v is chosen from

P' , and is chosen so as to be connected to R' , if $i = 1$, it follows immediately that R satisfies Properties 1 and 2.

Let $S' = N^{\vee, s}(R')$. By the induction hypothesis, $S' = P' \cup X'$. During the loop of Line 3, before the recursive call $\text{CsCLIQUESi}(R, P, X, s)$, we remove nodes from P' and add them to X' . Thus, when node v is chosen in Line 3, it still holds that $S' = P' \cup X'$. Let S be the set of nodes that are of distance at most s from all nodes in R' . Clearly, $S = S' \cap N^s(v)$. Now, since $P = P' \cap N^s(v)$ and $X = X' \cap N^s(v)$, it follows that $S = P \cup X$, as required in Property 3.

Finally, suppose that u_1, \dots, u_{k+1} is a prefix of $\omega_i(C)$ for some maximal connected s -clique C . This implies that also u_1, \dots, u_k is also a prefix of $\omega_i(C)$. Hence, $C - R' \subseteq P'$. Since we only choose, in Line 3, a node that is in P' (and hence, by Property 3 is of distance at most s from every node in R') and if $i = 1$, we only continue with nodes that are connected to R , it follows that no node in C is removed from P in Line 5. Otherwise, this would contradict the minimality of choice of u_{k+1} in the definition of $\omega_i(C)$. Hence, it follows that $C - R \subseteq P$, as required in Property 4.

Now, we show Property 5. Let (R'', P'', X'') be the lowest common ancestor of (R, P, X) and (R', P', X') . We consider two cases:

- Case 1: (R'', P'', X'') is the node (R, P, X) . (The case in which (R'', P'', X'') is the node (R', P', X') is identical.) In this case, Let v be the node on the outgoing edge of (R, P, X) on the path leading to (R', P', X') . Clearly, $v \notin R$ and $v \in R'$, and therefore $R \neq R'$.
- Case 2: Neither node among (R, P, X) and (R', P', X') is an ancestor of the other. Let v and v' be the nodes on the outgoing edges of (R'', P'', X'') on the paths leading to (R, P, X) and (R', P', X') , respectively. Suppose, without loss of generality, that the node v was chosen before v' in the loop of Line 3. Clearly, R contains v . However, since v is removed from the set P'' before the recursive call with v' , it follows that v is not in R' . Hence, $R \neq R'$.

Finally, we show Property 6. Let $\omega_i(C) = u_1, \dots, u_n$. We show, by induction, that for every prefix u_1, \dots, u_k of $\omega_i(C)$, it holds that u_1, \dots, u_k is a path in \mathcal{T}_i . Obviously, this holds for the empty prefix. Assume the required for a prefix of length k , and we show that the claim holds for a prefix of length $k + 1$. By the induction hypothesis, u_1, \dots, u_k is a path in \mathcal{T}_i leading to a node (R, P, X) . Observe that $R = \{u_1, \dots, u_k\}$. By Property 4, it holds that $C - R \subseteq P$, i.e., $u_{k+1} \in P$. For $i = 1$, by the definition of $\omega_i(C)$, it holds that u_{k+1} is connected to R . Hence, at some point, the node u_{k+1} will be chosen in Line 3. Therefore, there will be a recursive call made where u_{k+1} is added, i.e., u_1, \dots, u_{k+1} is a path in \mathcal{T}_i . \square

We can now show correctness of the algorithm.

THEOREM 5.4. *Let G be a graph. Then, calling the procedure $\text{CsCLIQUESi}(\emptyset, V(G), \emptyset, s)$ will result in the printing of every maximal connected s -clique in G precisely once, and no other sets of nodes will be printed.*

PROOF. First observe that no set will be printed more than once. This follows immediately from the Property 5 of Lemma 5.3, since we print the set R , and there are no two nodes in the execution tree that contain the same set R .

Next, we show that every set that is printed is a maximal s -clique. Assume that R is printed. Let (R, P, X) be the node of \mathcal{T}_i in which R is printed. By Property 1 of Lemma 5.3, it follows that R is an s -clique. If $i = 1$, R is also connected (Property 2 of

Lemma 5.3). If $i = 2$, R must be connected if it is printed, as this is part of the requirement before printing (Line 1). Hence, R is a connected s -clique. Suppose, by way of contradiction, that R is not maximal. Then there is a node v that is connected to R and is of distance at most s from every node in R . By Property 3 of Lemma 5.3, it holds that $v \in P \cup X$. Hence, either $P \cap N(R) \neq \emptyset$ or $X \cap N(R) \neq \emptyset$, and thus, R will not be printed, in contradiction to the assumption.

Finally, we show that every maximal connected s -clique will be printed. Let C be a maximal connected s -clique. By Property 6 of Lemma 5.3, it holds that $\omega_i(C)$ is a path in \mathcal{T}_i leading to a node (R, P, X) . Assume, by way of contradiction, that this node is not an output node. It then follows that there is a node $v \in (P \cap N(R)) \cup (X \cap N(R))$, i.e., v is connected to R and is in P or X . By Property 3 of Lemma 5.3, it follows that v is of distance at most s from every node in R . Hence, $R \cup \{v\}$ is a connected s -clique. However, $R = C$, and thus, this is a contradiction to the assumption that C is maximal. Hence, it follows that if C is a maximal connected s -clique, C will be printed during execution. \square

5.3 Optimizing the Algorithm

We consider two different strategies to optimize the algorithm CsCLIQUEs2 . Unfortunately, neither of these strategies are applicable to CsCLIQUEs1 , as will be made clear.

Pivoting. As discussed earlier, a critical improvement to CLIQUEs , which renders the algorithm efficient in practice, is that of pivoting [32]. This strategy reduces the branching factor in the execution tree by avoiding iteration over the entire set P , and iterating over $P - N(u)$, for some $u \in P \cup X$, instead. Node u is called the *pivot*. This technique can be applied when generating maximal connected s -cliques, due to the following proposition.

PROPOSITION 5.5. *Let C be a maximal connected s -clique and let $u \in V(G)$ be a node. Then, one of the following conditions must hold: (1) $u \in C$, (2) $C \not\subseteq N^s(u)$ or (3) $C \cap N(u) = \emptyset$.*

PROOF. Suppose, by way of contradiction, that none of the above conditions hold. Then,

- (1) $u \notin C$;
- (2) $C \subseteq N^s(u)$;
- (3) C contains a node in $N(u)$.

It immediately follows that C is not maximal, in contradiction to the assumption, as u can be added to C (since it is close enough to all nodes in C , and has a neighbor in C). \square

It may not be immediately obvious how to utilize this property, to improve the runtime, as it states that every clique must either contain u , or contain some node that is not at distance s from u , or cannot contain any neighbor of u . Thus, if instead of iterating over P , we iterate over $P - N^s(u)$ (in a similar fashion to the pivoting of CLIQUE), we can miss maximal connected s -cliques that do not contain any node from $P - N^s(u)$ (and also contain no neighbor of u).

To overcome this problem, we will always choose the pivot as a neighbor of some node in R . To be precise, we choose a pivot node $u \in (P \cup X) \cap N^{\exists, 1}(R)$ that minimizes $P - N^s(u)$. Then, in Line 3 of CsCLIQUEs2 , instead of iterating over P , we iterate over $P - N^s(u)$. Now, every maximal connected s -clique containing R must either contain u , or must contain some node that is not within $N^s(u)$. (Otherwise, if it does not contain any node in $N^s(u)$, we could always add u , since u is connected to R and of distance at most s from every node in R .)

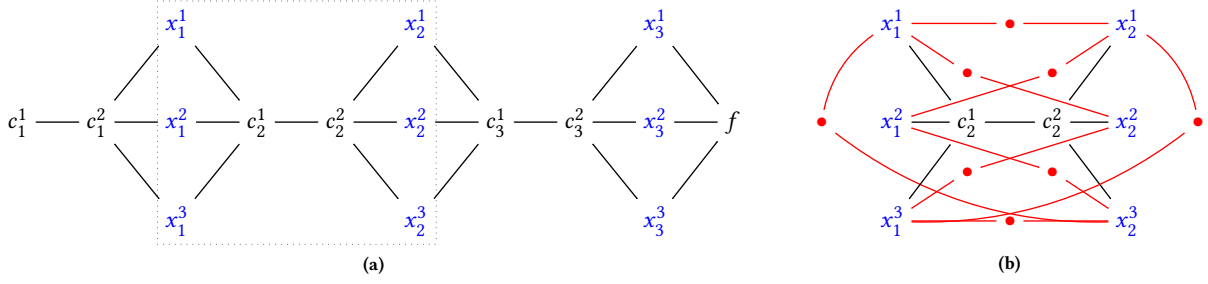


Figure 8: Graph used to demonstrate the reduction of Theorem 5.6

We note that this improvement cannot be integrated with CsCLIQUEs1, as it requires us to iterate over nodes that may not be neighbors of R . The algorithm CsCLIQUEs1 always preserves the invariant that R is connected, and iterating over such nodes would cause them to be added to R , thereby, losing the correctness of the invariant.

To summarize, integrating pivoting into CsCLIQUEs2 is performed by replacing Line 3 with the following two lines:

- 3.1 $u \leftarrow \arg \min_u \{|P - N^s(u)| \mid u \in (P \cup X) \cap N^{\exists 1}(R)\}$
 3.2 **for** $v \in P - N^s(u)$

Checking for Feasibility. CsCLIQUEs1 never creates a set R that is unconnected. On the other hand, CsCLIQUEs2 may go deep into the recursion tree even if a set R is being considered, for which it is not possible to add nodes and derive a connected maximal s -clique. It would be preferable to prune branches for calls of the form (R, P, X, s) if there is no connected s -clique C such that $R \subseteq C$ and $C \subseteq R \cup P$. (Recall that we will only try to add nodes from P to R .) In such cases, the branches cannot lead to a solution. Unfortunately, this cannot be verified efficiently, in the general case, as shown stated by the following theorem.

THEOREM 5.6. *Let G be a graph, $s > 1$ be an integer, and R be an s -clique. Determining whether there exists a connected s -clique C such that $R \subseteq C$ is NP-complete.*

PROOF. Membership in NP is immediate, as we can guess a set C and check whether it satisfies all requirements. We show NP-hardness by a reduction from 3-SAT.

Let ψ be a 3-SAT formula with m clauses C_1, \dots, C_m , over the variables X_1, \dots, X_k . We assume, without loss of generality, that no clause contains both a variable and its negation.

Let V_0 be the set of nodes:

$$\{c_i^k \mid i \leq m, k \leq s\} \cup \{x_i^j \mid i \leq m, j \leq 3\} \cup \{f\}$$

Let G_0 be the graph derived by taking the nodes in V_0 , and adding the following edges:

- $\{c_i^j, c_i^{j+1}\}$ for every $i \leq m$ and $j < s$;
- $\{c_i^s, x_i^j\}$ for every $i \leq m$ and $j \leq 3$;
- $\{x_i^j, c_{i+1}^1\}$ for every $i < m$ and $j \leq 3$;
- $\{x_m^j, f\}$ for every $j \leq 3$.

We say that a pair of nodes in V_0 is *conflicting* if they are of the form $x_i^j, x_{i'}^{j'}$ and the j -th literal in C_i is the negation of the j' -th literal in $C_{i'}$. Let G be the graph derived by taking the graph G_0 , and adding a path of length s (using new nodes) between every pair of non-conflicting nodes in V_0 that are at distance greater than s in G_0 .

Figure 8 (a) demonstrates graph G_0 for $s = 2, m = 3$ and the 3-SAT formula $\psi = (X_1 \vee \bar{X}_2 \vee X_3) \wedge (X_1 \vee X_2 \vee X_3) \wedge (\bar{X}_1 \vee \bar{X}_2 \vee X_3)$. Figure 8 (b) focuses on the framed subgraph in (a) and shows all the nodes and edges that will appear in G . Observe that there is no path of length $s = 2$ between x_1^2 and x_2^2 as they correspond to \bar{X}_2 and X_2 .

Now, we make several observations about the graph G .

- (1) Let U be any subset of V_0 that does not contain nodes corresponding to a literal and its negation. Then, U is an s -clique (although may be unconnected). This is immediate, since we have paths of length s between every two nodes in U .
- (2) Let U be any subset of $V(G)$ that does contain nodes corresponding to a literal and its negation. Then, U is not an s -clique. This is also apparent, from a careful analysis of the graph. We did not include paths of length s between such nodes, in the beginning. Furthermore, when adding edges between nodes in V_0 , we never create a path of length at most s between such nodes. Indeed, the use of s nodes c_1^1, \dots, c_i^s is precisely to avoid such cases.
- (3) Let U be any subset of $V(G)$ containing the nodes c_1^1, \dots, c_m^1, f . If U is an s -clique, then U must be a subset of V_0 , i.e., cannot contain the additional new nodes. To see why this is so, observe that every node that is not among V_0 is on a path of length s between two nodes in V_0 , and will be at distance greater than s from at least one node among c_1^1, \dots, c_m^1, f .

Now, consider s -clique $R = \{c_1^1, \dots, c_1^s, \dots, c_m^1, \dots, c_m^s, f\}$. We claim that there exists a connected s -clique C such that $R \subseteq C$ if and only if ψ is satisfiable. Suppose first that ψ is satisfiable, and let μ be a satisfying assignment for ψ . Let C be the set containing R , as well as all nodes corresponding to literals that are satisfied in μ . Clearly, C is an s -clique, by our first observation. In addition, since C must include at least one node x_i^j corresponding to a satisfied literal in each clause c_i , the set C is connected (as x_i^j will connect between c_i^s and c_{i+1}^1).

For the other direction, suppose there exists such a set C . By Observation 3, C can contain only nodes from V_0 , and by Observation 2, C will not contain nodes corresponding to a literal and its negation. Therefore, C defines a truth assignment for ψ , in which true is assigned to literals corresponding to nodes in C . As before, the fact that C is connected implies that the truth assignment gives the value of true to at least one literal in each C_i , i.e., ψ is satisfiable. \square

Theorem 5.6 implies that we cannot have an efficient algorithm that prunes useless branches whenever possible. Instead, we

apply a simple optimization, that is sufficient, but not complete, for pruning (i.e., branches are pruned only if they cannot lead to a result, but not all such branches will be pruned). In particular, we remove from P each node v for which the set of nodes $R \cup \{v\}$ is not completely contained in a single connected component of

$$G[R \cup \{v\} \cup (P \cap N^s(v))].$$

Intuitively, v can be added to R , only if eventually, we may find nodes in $P \cap N^s(v)$ ($= N^{v,s}(R \cup \{v\})$) that can fill in the gaps between the nodes in R to derive a single connected graph. Therefore, if we discover that this is not the case, we can remove v from P , as it can never be in a connected s -clique together with R .

Example 5.7. Consider running CsCLIQUES2 algorithm with the above feasibility check, on graph H from Figure 3 (a). When first called, $R = X = \emptyset$, $P = \{v_1, \dots, v_6\}$ and $s = 2$. Since, R is empty, we iterate over all nodes in P . Suppose the first node chosen is v_1 . In the recursive call to CsCLIQUES2, we will have $R = \{v_1\}$, $P = \{v_2, v_3, v_5, v_6\}$ and $X = \emptyset$. We will iterate over all nodes in P . In the second iteration of the loop of Lines 3-6, $v = v_3$ (v_2 was removed from P at the end of the first iteration) and $P \cap N^s(v) = v_5$. Now $R \cup \{v\} = \{v_1, v_3\}$ is not a connected component in $G[R \cup \{v\} \cup (P \cap N^s(v))]$ $= \{v_1, v_3, v_5\}$ and v_3 will be removed from P without calling the recursion.

6 FINDING LARGE RESULTS

We now consider the problem of finding large maximal connected s -cliques. In particular, assume we are given an integer $k \geq 0$, and our goal is to find all maximal connected s -cliques C such that $|C| \geq k$. When $s = 1$, s -cliques are standard cliques. For this case, it is well known that determining whether there exists a clique of size k is NP-complete. Therefore, it is interesting to consider k as a parameter of the problem and determine whether a fixed parameter algorithm exists, i.e., whether there is an algorithm that runs in time $O(f(k) \cdot |G|^{O(1)})$, for an arbitrary function f . Since k is expected to be much smaller than $|G|$, such an algorithm would be useful. Unfortunately, determining existence of a clique is known to be $W[1]$ -complete, i.e., we cannot expect to find an algorithm with time $O(f(k) \cdot n^{O(1)})$.

We now consider the case in which $s > 1$. Interestingly, determining whether there exists a s -clique of size k is fixed parameter tractable with respect to k , for both the case of connected s -cliques, and for arbitrary s -cliques. This gives hope that large maximal connected s -cliques can be enumerated with delay between answers that is exponential in k , but not in $|G|$. Unfortunately, for general s -cliques, this is not the case, as the following theorem states. (The proof has been omitted due to space limitations.) For connected s -cliques this problem is still open.

THEOREM 6.1. *Let k and $s > 1$ be integers.*

- (1) *The problem of determining whether a graph G contains a (connected) s -clique of size k is NP-complete, but is fixed parameter tractable with respect to k .*
- (2) *It is not possible to enumerate all maximal s -cliques of size at least k in a graph G with fixed parameter delay with respect to k , unless $W[1] = W[0]$.*

Our algorithms can already mine all maximal connected s -cliques of size at least k , by simply finding all maximal connected s -cliques, and then filtering out all those that do not satisfy the size bound. This process may be highly inefficient, as many smaller s -cliques will be generated. We consider optimizations that can be made to our algorithms to speed up the process

of finding all maximal connected s -cliques of size at least k . In POLYDELAYENUM, we replace the queue with a priority queue that returns larger maximal connected s -cliques first. In the algorithms CsCLIQUES1 and CsCLIQUES2, we prune (i.e., do not make recursive calls) the cases in which $|R| + |P| < k$. Clearly, in such cases it is not possible to create maximal connected s -cliques with k nodes at least.

7 EXPERIMENTAL RESULTS

Our algorithms were implemented in 32bit C++, as an extension of the SNAP library [21]. All experimentation was run on a Win7 desktop with 16GB RAM and an Intel i5-4570 processor, with 2GB of memory allocated to the program.

We run our algorithms on synthetic Erdős-Rényi (ER) graphs and scale-free (SF) graphs (which simulate social networks) of varying sizes, generated by the SNAP library. All data points in our figures are derived by generating three random graphs of the same size, and taking the average runtime in seconds. We also use several real datasets, as follows, taken from [20]. (Some of these datasets are directed graphs, but for our experimentation, we ignore the direction of the edges.)

- DBLP, with 317,080 nodes and 1,049,866 edges
- Amazon, with 334,863 nodes and 925,872 edges
- LiveJournal, with 3,997,962 nodes and 34,681,189 edges
- Twitter, with 81,306 nodes and 1,768,149 edges
- YouTube, with 1,134,890 nodes and 2,987,624 edges

One of the most costly operations in all algorithms is computing the set $N^s(v)$ for various nodes v . To save time, whenever we compute this set, we store it in a hash table, to be reused again later on, if needed. For large data sets, there is insufficient memory to store all neighbors of distance s within the hash table. When memory begins to run low, we remove some entries from the hash table (using an LRI ordering) to make room for new neighbor results.

Comparing Bron-Kerbosch Adaptations. We start by comparing the baseline versions of our Bron-Kerbosch adaptations, i.e., CSCLIQUE1 and CSCLIQUE2, with the versions including the two optimizations considered. We append the letter ‘‘P’’ and ‘‘F’’ to CSCLIQUE2 to indicate that pivoting is used and that our feasibility check is performed.

Figure 9a shows the result of running our adaptations of the Bron-Kerbosch algorithm to find 100 connected 2-cliques, with and without various optimizations, for random ER graphs of varying numbers of nodes (between one thousand and one million), and average node degree 10. As was to be expected, CSCLIQUE1 is significantly superior to CSCLIQUE2. However, all three versions that do not use pivoting are much slower than the pivoting versions (the two overlapping lines at the bottom of the graph), running 30 to 60 times slower than the pivoting versions, for graphs of one million nodes. For SF graphs, the gap is even larger, with the non-pivoting versions running approximately 120 times slower on graphs with only one thousand nodes (and average node degree 10). For this reason, in the remainder of the experimentation, we only consider the algorithms CSCLIQUE2P and CSCLIQUE2PF, along with POLYDELAYENUM (which will be denoted PD in our graphs).

Varying Node Size. We continue in our study of how the size of the graph (as determined by the number of its nodes) affects runtime. As before, we generated random graphs with between

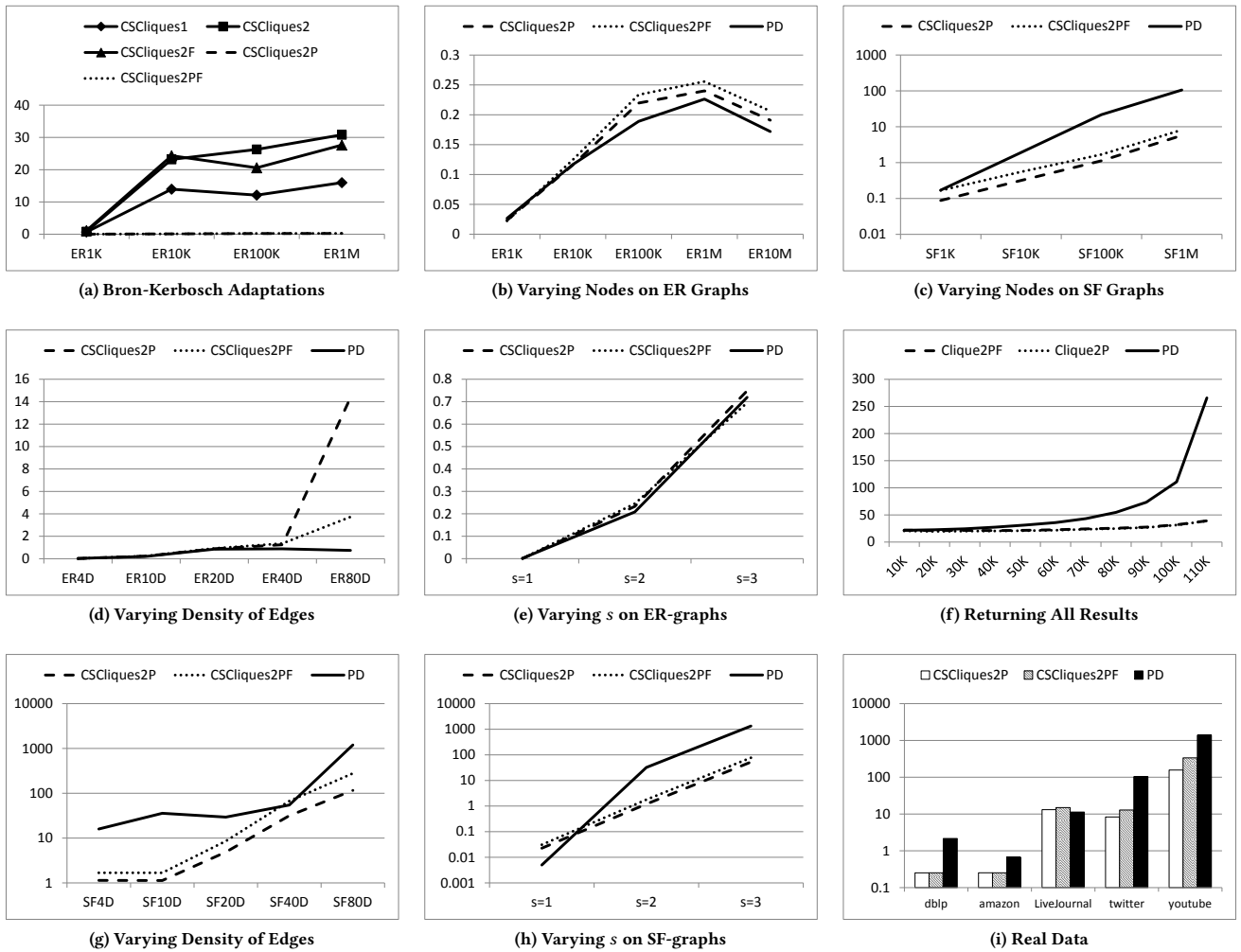


Figure 9: Execution times for varying parameters.

one thousand and one million nodes, and average node degree 10. We chose $s = 2$ and measured the time to return 100 connected s -cliques.

This experiment appears in Figures 9b and 9c. All times are in seconds, and that in Figure 9c the times are in log-scale. Algorithm POLYDELAYENUM (PD in the graphs) slightly outperforms CSCLIQUES2P and CSCLIQUES2PF on the ER graphs, but is significantly worse on the SF graphs. In addition, all algorithms perform worse on the latter, than on the former, probably due to the fact that average size of connected 2-cliques generated is much larger in a scale-free graph. For example, POLYDELAYENUM returned connected 2-cliques of average size 10.79 and 105.83, respectively, on the ER and SF graphs of size one million. The change in the trend of Figure 9b between 1M and 10M follows since the average size of connected 2-cliques is smaller over graphs with 10 million nodes, as the graph is sparser.

Varying Edge Density. We consider how the density of the edges affects the speed in which results can be returned. We generated random graphs with 100 thousand nodes, and an average degree of 4, 10, 20, 40 and 80. We chose $s = 2$, and measure the time to return 100 connected 2-cliques.

Figures 9d and 9g contain the result of this experiment for ER and SF graphs, respectively. As the density increases, algorithm POLYDELAYENUM once again outperforms CSCLIQUES2P and CSCLIQUES2PF on the ER graphs. CSCLIQUES2PF is superior to CSCLIQUES2P for degree density 80, as it avoids many recursive calls. (This is in contrast to the many other cases in which we observe that its time is inferior, as the overhead for testing feasibility is large.) On the SF graphs, on the other hand, POLYDELAYENUM is the slowest and CSCLIQUES2PF performs slightly worse than CSCLIQUES2P probably do to the high connectivity of scale-free graphs, which causes the feasibility check to always return true.

Varying s . We study how the choice of s affects the runtime. Once again, we generated a random graph with 100K nodes and average degree of 10. We measure the time to generate 100 connected s -cliques for values of s varying from 1 to 3. Recall that when $s = 1$, we actually return cliques.

The runtime appears in Figures 9e and 9h. Runtime increases as s increases, as it is increasingly more expensive to find neighbors of distance s . As before, runtime is significantly slower for SF graphs, as they have s -cliques that are much larger. The algorithm POLYDELAYENUM is slower than the others, but returns s -cliques that are larger, on average. (For example, for $s = 3$, on SF graphs,

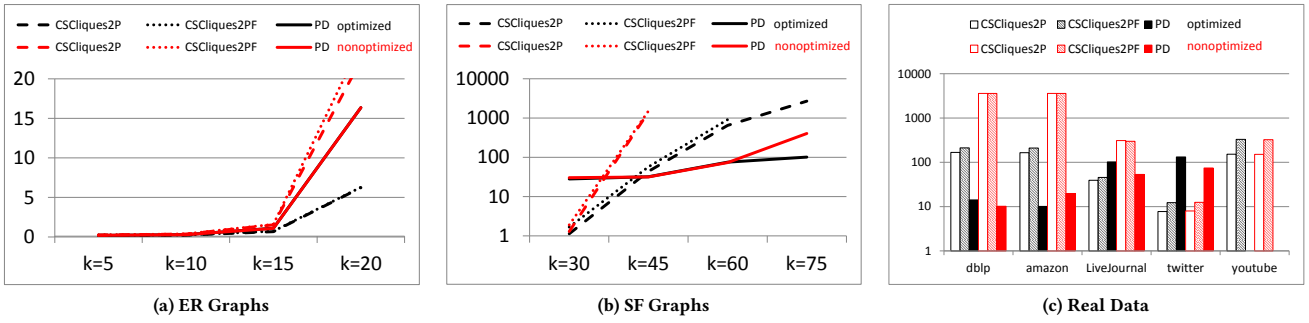


Figure 10: Execution time for returning large results.

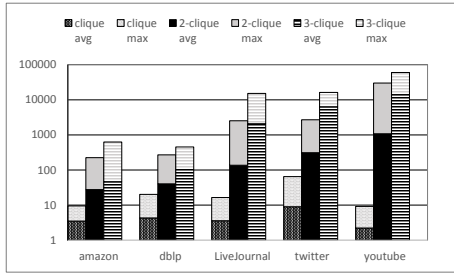


Figure 11: Average and max sizes.

POLYDELAYENUM returns 3-cliques of average size 1084, while the other algorithms return 3-cliques with average size 668.) Even for $s = 3$, the runtime remains reasonable, and in practice, larger values of s are usually not of interest.

Returning all Results. We consider the runtime when returning *all* connected s -cliques in a graph. For this experiment, we generated an ER graph with 100 thousand nodes and average degree of 10. This graph has 112,134 connected 2-cliques. We measure time elapsed between generating every 10 thousand results, until all results are generated.

Figure 9f contains the result of this experiment. Recall that POLYDELAYENUM runs in polynomial delay (Theorem 4.2), i.e., the delay between results is polynomial in $|V(G)|$. However, the CSCLIQUES s variations have no such guarantee. It is therefore somewhat surprising to see that, in practice, the delay between successive sets of 10 thousand results grows for POLYDELAYENUM while for CSCLIQUES2P and CSCLIQUES2PF remaining almost steady. Perhaps this can be explained by the high memory requirements of POLYDELAYENUM (e.g., as it must store all results created), which makes access to auxiliary data structures more expensive as more results are created.²

Non-synthetic Datasets. In Figure 9i we show the results of returning 100 connected 2-cliques over the real datasets presented in the beginning of this section. Once again, the algorithms CSCLIQUES2P and CSCLIQUES2PF outperform POLYDELAYENUM almost consistently. POLYDELAYENUM performs better only over the LiveJournal dataset, but the difference in runtime is small.

Returning Large Results. We consider the heuristics presented in Section 6 to find 100 connected 2-cliques that are larger than k ,

²Also, the reader may note that while the delay given by Theorem 4.2 is polynomial in the input, it is still quite large.

for some given size k . Once again, we generated random graphs with 100K nodes and average degree of 10 and used the real datasets as well. Our range of values for k for the random graphs was determined by the average size of answers returned by the algorithms when no size restriction was given. (With no size restriction, the average result size for the ER graph was approximately 10, and for the SF graph was between 35 and 60, depending on the algorithm used.) For the real graphs, we set $k = 10$.

In Figures 10a, 10b and 10c, the result of this experiment appears, comparing the optimized version of the algorithms (in black) against the regular algorithms (in red) until a hundred large results are found. Interestingly, the time for POLYDELAYENUM on SF graphs is steady, as this algorithm naturally creates large results in this setting. The Bron-Kerbosch adaptations timed out on SF graphs with large k and POLYDELAYENUM timed out on the youtube graph, running over an hour. Clearly, the optimizations for CSCLIQUES2P and CSCLIQUES2PF improve the runtime significantly in most cases, but the optimization for algorithm POLYDELAYENUM is not consistently superior.

We note that the optimizations discussed in Section 6 were very important in achieving the runtimes shown. The speedup with respect to the regular (non-optimized) versions were significant in most cases.

Comparing cliques and s -cliques size. To further motivate enumerating s -cliques for $s > 1$, we compare the sizes of s -cliques for $s = 1, 2, 3$ over the real datasets. For this experiment we randomly sampled 100 s -cliques, for each of the datasets and each of the values of s . We then calculated both average and maximum sizes. The result of this experiment appears in Figure 11. The datasets are organized by increasing edge density, and for each graph, we plot the average and maximum size of cliques, 2-cliques and 3-cliques. Unsurprisingly, the size of s -cliques is larger (for all choices of s) when the graph is more dense. In addition, observe that as s grows larger, the average and maximum size of the s -cliques increases. Note that the sizes are in log scale, and hence, differ significantly.

Depending on the application, finding highly cohesive sets of larger (or smaller) sizes may be more useful. For example, in many settings, communities that are very small, or very large, may be less useful. Smaller communities may not well-represent the actual facts on the ground, while huge communities may contain people who are not sufficiently related. The ability to choose a value for s , and then enumerate (as our algorithms do) gives the user maximum flexibility.

Summary of Results. As is apparent from the results above, all algorithms POLYDELAYENUM, CsCLIQUES2P and CsCLIQUES2PF have runtimes that are quite reasonable. Usually, the adaptations of Bron-Kerbosch have superior runtime to that of the algorithm POLYDELAYENUM, but the latter may be preferable for returning larger s -cliques, particularly on sparse graphs. When comparing CsCLIQUES2P and CsCLIQUES2PF, one can observe that usually the overhead of checking for feasibility (done in the latter algorithm) is larger than the gains derived by avoiding unnecessary recursive calls. (The only exception was for highly dense ER-graphs, in which feasibility checking clearly pays off.)

Another important difference between POLYDELAYENUM and the Bron-Kerbosch adaptations is in memory requirements. Running POLYDELAYENUM requires additional data structures (e.g., the queue Q and index I). Thus, its memory requirements are linear in the size of the output. This contrasts with the Bron-Kerbosch adaptations, which require memory that is linear in the input. In our implementation, we assumed that the structures for POLYDELAYENUM fit in main memory, but for larger inputs (or when desiring to run the algorithm until all results have been found), it would be necessary to use external memory structures.

8 CONCLUSION

This paper studied the problem of finding maximal connected s -cliques in a graph—a problem of high interest, due to the usefulness of clique relaxations. We have presented the first algorithms for this problem, by taking two completely different approaches for solving this problem. The correctness of our algorithms is proven and experimentation shows the efficiency of our approaches.

As future work, we intend to study applications in which connected s -cliques can be useful, such as community detection and link prediction. Optimizations of the algorithms, for special types of graphs (e.g., sparse graphs or bipartite graphs) are also an interesting direction for future work. Another important direction is adapting the algorithms to a distributed environment, as returning all s -cliques for large graphs can become infeasible for a single machine.

ACKNOWLEDGMENTS

The authors were partially supported by the Israel Science Foundation (Grant 879/16).

REFERENCES

- [1] E. A. Akkoyunlu. 1973. The Enumeration of Maximal Cliques of Large Graphs. *SIAM J. Comput.* 2, 1 (1973), 1–6. <https://doi.org/10.1137/0202001>
- [2] Balabhaskar Balasundaram, Sergiy Butenko, and Svyatoslav Trukhanov. 2005. Novel Approaches for Analyzing Biological Networks. *J. Comb. Optim.* 10, 1 (2005), 23–39. <https://doi.org/10.1007/s10878-005-1857-x>
- [3] Devora Berlowitz, Sara Cohen, and Benny Kimelfeld. 2015. Efficient Enumeration of Maximal k -Plexes. In *SIGMOD*.
- [4] Kamanashis Biswas, Vallipuram Muthukkumarasamy, and Elankayer Sithirasanen. 2013. Maximal clique based clustering scheme for wireless sensor networks. In *ISSNIP*.
- [5] Vladimir Boginski, Sergiy Butenko, and Panos M. Pardalos. 2005. Statistical analysis of financial networks. *Comput. Statistics and Data Analysis* 48, 2 (2005), 431–443. <https://doi.org/10.1016/j.csda.2004.02.004>
- [6] Coen Bron and Joep Kerbosch. 1973. Algorithm 457: Finding All Cliques of an Undirected Graph. *Commun. ACM* 16, 9 (Sept. 1973), 575–577. <https://doi.org/10.1145/362342.362367>
- [7] Lijun Chang, Jeffrey Xu Yu, and Lu Qin. 2013. Fast Maximal Cliques Enumeration in Sparse Graphs. *Algorithmica* 66, 1 (2013), 173–186. <https://doi.org/10.1007/s00453-012-9632-8>
- [8] James Cheng, Yiping Ke, Ada Wai-Chee Fu, Jeffrey Xu Yu, and Linhong Zhu. 2011. Finding maximal cliques in massive networks. *ACM Trans. Database Syst.* 36, 4 (2011), 21:1–21:34. <https://doi.org/10.1145/2043652.2043654>
- [9] James Cheng, Yiping Ke, Ada Wai-Chee Fu, Jeffrey Xu Yu, and Linhong Zhu. 2011. Finding Maximal Cliques in Massive Networks. *ACM Trans. Database Syst.* 36, 4, Article 21 (Dec. 2011), 34 pages. <https://doi.org/10.1145/2043652.2043654>
- [10] Sara Cohen, Benny Kimelfeld, and Yehoshua Sagiv. 2008. Generating all maximal induced subgraphs for hereditary and connected-hereditary graph properties. *J. Comput. Syst. Sci.* 74, 7 (2008), 1147–1159.
- [11] Alessio Conte, Roberto De Virgilio, Antonio Maccioni, Maurizio Patrignani, and Riccardo Torlone. 2016. Finding All Maximal Cliques in Very Large Social Networks. In *Proceedings of the 19th International Conference on Extending Database Technology, EDBT 2016, Bordeaux, France, March 15–16, 2016, Bordeaux, France, March 15–16, 2016*. 173–184. <https://doi.org/10.5441/002/edbt.2016.18>
- [12] David Eppstein, Maarten Löffler, and Darren Strash. 2013. Listing All Maximal Cliques in Large Sparse Real-World Graphs. *ACM Journal of Experimental Algorithmics* 18 (2013). <https://doi.org/10.1145/2543629>
- [13] Y. Fried, D.A. Kessler, and N.M. Shnerb. 2016. Communities as Cliques. *Nature Scientific Reports* 6, 35648 (2016).
- [14] E. Harley, A. Bonner, and N. Goodman. 2001. Uniform integration of genome mapping data using intersection graphs. *Bioinformatics* 17, 6 (2001), 487–494.
- [15] Hiro Ito and Kazuo Iwama. 2009. Enumeration of Isolated Cliques and Pseudo-cliques. *ACM Trans. Algorithms* 5, 4, Article 40 (Nov. 2009), 21 pages. <https://doi.org/10.1145/1597036.1597044>
- [16] Daxin Jiang and Jian Pei. 2009. Mining Frequent Cross-graph Quasi-cliques. *ACM Trans. Knowl. Discov. Data* 2, 4, Article 16 (Jan. 2009), 42 pages. <https://doi.org/10.1145/1460797.1460799>
- [17] David S. Johnson, Christos H. Papadimitriou, and Mihalis Yannakakis. 1988. On Generating All Maximal Independent Sets. *Inf. Process. Lett.* 27, 3 (1988), 119–123.
- [18] Mehdi Kargar and Aijun An. 2011. Keyword Search in Graphs: Finding R-cliques. *Proc. VLDB Endow.* 4, 10 (July 2011), 681–692. <https://doi.org/10.14778/2021017.2021025>
- [19] Ina Koch. 2001. Enumerating All Connected Maximal Common Subgraphs in Two Graphs. *Theor. Comput. Sci.* 250, 1-2 (Jan. 2001), 1–30. [https://doi.org/10.1016/S0304-3975\(00\)00286-3](https://doi.org/10.1016/S0304-3975(00)00286-3)
- [20] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. <http://snap.stanford.edu/data>. (June 2014).
- [21] Jure Leskovec and Rok Sosič. 2014. SNAP: A general purpose network analysis and graph mining library in C++. <http://snap.stanford.edu/snap>. (June 2014).
- [22] David Liben-Nowell and Jon M. Kleinberg. 2007. The link-prediction problem for social networks. *JASIST* 58, 7 (2007), 1019–1031. <https://doi.org/10.1002/asi.20591>
- [23] Guimei Liu and Limsoon Wong. 2008. Effective Pruning Techniques for Mining Quasi-Cliques. In *ECML PKDD*.
- [24] R.Duncan Luce. 1950. Connectivity and generalized cliques in sociometric group structure. *Psychometrika* 15, 2 (1950), 169–190. <https://doi.org/10.1007/BF02289199>
- [25] S. Mohseni-Zadeh, P. Brézellec, and J.-L. Risler. 2004. Cluster-C, an algorithm for the large-scale clustering of protein sequences based on the extraction of maximal cliques. *Comput. Biology and Chemistry* 28, 3 (2004), 211 – 218. <https://doi.org/10.1016/j.compbiolchem.2004.03.002>
- [26] J.W. Moon and L. Moser. 1965. On cliques in graphs. *Israel Journal of Mathematics* 3, 1 (1965), 23–28. <https://doi.org/10.1007/BF02760024>
- [27] Kevin A. Naudé. 2016. Refined pivot selection for maximal clique enumeration in graphs. *Theoretical Computer Science* 613 (2016), 28 – 37.
- [28] F. Mahdavi Pajouh and B. Balasundaram. 2012. On inclusionwise maximal and maximum cardinality k -clubs in graphs. *Discrete Optimization* 9, 2 (2012), 84–97. <https://doi.org/10.1016/j.disopt.2012.02.002>
- [29] G. Palla, I. Derenyi, and T. Vicsek. 2005. Uncovering the overlapping community structure of complex networks in nature and society. *Nature* 435, 7043 (2005), 814–818.
- [30] Jeffrey Pattillo, Nataly Youssef, and Sergiy Butenko. 2013. On clique relaxation models in network analysis. *European Journal of Operational Research* 226, 1 (2013), 9–18. <https://doi.org/10.1016/j.ejor.2012.10.021>
- [31] Lu Qin, Rong-Hua Li, Lijun Chang, and Chengqi Zhang. 2015. Locally Densest Subgraph Discovery. In *KDD*.
- [32] Etsuji Tomita, Akira Tanaka, and Haruhisa Takahashi. 2006. The Worst-case Time Complexity for Generating All Maximal Cliques and Computational Experiments. *Theor. Comput. Sci.* 363, 1 (Oct. 2006), 28–42. <https://doi.org/10.1016/j.tcs.2006.06.015>
- [33] Charalampos Tsourakakis, Francesco Bonchi, Aristides Gionis, Francesco Gullo, and Maria Tsiarli. 2013. Denser Than the Densest Subgraph: Extracting Optimal Quasi-cliques with Quality Guarantees. In *SIGKDD*.
- [34] Stanley Wasserman and Katherine Faust. 1994. *Social Network Analysis: Methods and Applications*. Cambridge University Press.
- [35] Bin Wu and Xin Pei. 2007. A Parallel Algorithm for Enumerating All the Maximal k -Plexes. In *PAKDD*.
- [36] B. Yan and S. Gregory. 2009. Detecting communities in networks by merging cliques. In *ICIS*.
- [37] Zhiping Zeng, Jianyong Wang, Lizhu Zhou, and George Karypis. 2007. Out-of-core Coherent Closed Quasi-clique Mining from Large Dense Graph Databases. *ACM Trans. Database Syst.* 32, 2 (June 2007).
- [38] Zhaonian Zou, Jianzhong Li, Hong Gao, and Shuo Zhang. 2010. Finding top- k maximal cliques in an uncertain graph. In *ICDE*.

Summarization Algorithms for Record Linkage

Dimitrios Karapiperis
Hellenic Open University
Patras, Greece
dkarapiperis@eap.gr

Aris Gkoulalas-Divanis
IBM Watson Health
Cambridge, MA, USA
gkoulala@us.ibm.com

Vassilios S. Verykios
Hellenic Open University
Patras, Greece
verykios@eap.gr

ABSTRACT

Record linkage has received significant attention in recent years due to the plethora of data sources that have to be integrated to facilitate data analyses. In several cases, such an integration involves disparate data sources containing huge volumes of records and must be performed in near real-time in order to support critical applications. In this paper, we propose the first summarization algorithms for speeding up *online* record linkage tasks. Our first method, called SkipBloom, summarizes efficiently the participating data sets, using their blocking keys, to allow for very fast comparisons among them. The second method, called BlockSketch, summarizes a block to achieve a constant number of comparisons for a submitted query record, during the matching phase. Additionally, we extend BlockSketch to adapt its functionality to streaming data, where the objective is to use a constant amount of main memory to handle potentially unbounded data sets. Through extensive experimental evaluation, using three real-world data sets, we demonstrate the superiority of our methods against two state-of-the-art algorithms for online record linkage.

1 INTRODUCTION

Massive amounts of data, stored in disparate data sources, have to be integrated and matched to support data analyses that can be highly beneficial to businesses, governments, and academia. Record linkage, also known as *entity resolution* or *data matching*, is the process of identifying records that *match*, i.e., refer to the same real-world entity. The lack of common unique identifiers for records that belong to different data sources, as well as the existence of variations, errors, misspellings, and typos in various data fields, constitute record linkage a challenging process. Traditionally, record linkage consists of two main steps: *blocking* and *matching*. In the blocking step, records that potentially match are grouped into the same block. Subsequently, in the matching step, records that have been blocked together are examined to identify those that match. Matching is implemented using either a *distance function*, which compares the respective field values of a record pair against specified distance thresholds, or a *rule-based approach*, e.g., “if the surnames and the zip codes match, then classify the record pair as matching”.

Several blocking approaches have been developed with the aim to scale the record linkage process to Big data sets without sacrificing accuracy [1, 6, 14, 32]. These methods perform the linkage process offline and provide the result set only when the entire linkage process has been completed. Given the size of modern data sets and the costly operations that have to be performed for record linkage, offline methods can take a significant amount of time to produce the matchings. There are many cases though, where *the linkage process has to return a fast response in order*

to allow for emergency actions to be triggered. Let us assume, for example, a central crime detection system that collects data from several sources, such as crime and immigration records, central citizens’ repositories, and airline transactions. Query data about a suspect could be submitted to this system in order to be matched with any possible similar records found therein. The results of this process have to be reported as fast as possible or, at least, within an acceptably low time period, in order to trigger police enforcement actions.

As another example, consider the recent series of bank and insurance company failures, which triggered a financial crisis of unprecedented severity. In order for these institutions to recover and return to normal business operation, they had to engage in merger talks. One of the driving forces of such mergers is the appreciation of the extent to which the customer databases of the constituent institutions are shared, so that the benefits of the merger can be proactively assessed in a timely manner. A very fast estimation of the extent of the overlap of the customer databases is thus a decisive factor in the merger process. To achieve this, the data custodians could use *summaries of their databases* in order to quickly estimate the overlap of their customers, instead of engaging in a tedious record linkage task. Although our motivation comes from the summarization of the blocking structure of a database, we believe that database summarization is an area of great interest with applications beyond record linkage.

To support real-world applications where record linkage has to be performed in near real-time, several online record linkage approaches have been proposed in the literature [5, 10, 24, 31]. These approaches require the availability of large amounts of main memory, which is necessary in order to store their corresponding data structures. For instance, [5] utilizes large inverted indexes, while [10, 24, 31] sort the records to form blocks by leveraging large matrices or huge graphs. Despite several efforts to utilize small amounts of memory, e.g., [24], the results in terms of performance clearly indicate the *inability of these algorithms to handle an increasingly large volume (or a continuous stream) of records in a real-time fashion*. Given that main memory is always *bounded* and the number of records may in several real-world applications be *unbounded*, the performance of these data structures quickly degrades significantly. Furthermore, in order to deal with this plethora of records and detect the matching pairs, the proposed methods usually resort to conducting *an excessive number of distance computations*. This strategy, however, is not efficient, since it incurs significant delays to the record linkage process.

In this paper, we introduce three methods for efficiently managing large volumes of records in the context of online record linkage. Our first method, called SkipBloom, performs a summarization (synopsis) of the blocking structure of a data set using a small footprint of main memory, whose size is logarithmic in the number of distinct processed blocking keys. This synopsis can be easily transferred to another site (or used remotely) to estimate the common number of blocking keys. Such a preliminary estimation may bring to surface important insights, which can

be further analyzed by the data custodians. The outcome of such analyses may encourage (or discourage) the data custodians to conduct a full-scale record linkage task.

Our second method, called BlockSketch, tackles the problem of blocks that are overwhelmed with records, which should be compared against a query record to detect matching pairs. BlockSketch instead of implementing the naïve linear approach, compares the query record with a *constant number of records in the target block*, which entails a bounded matching time. In order to achieve this optimization, BlockSketch compiles, for each block, a number of sub-blocks, which reflect the distances of the underlying blocked records from the blocking key. The algorithm places a query record to the sub-block whose records exhibit the smallest distances from the query record.

Our third method, called SBlockSketch, operates on data streams, where the entire data set is not known a-priori but, instead, there is an unbounded stream of incoming data records. SBlockSketch maintains a constant number of blocks in main memory at the cost of a time overhead during their replacement with blocks that reside in secondary storage. In this scheme, we propose a selection algorithm to effectively select the blocks that should be replaced, by taking into account their selectivity (by the incoming records) and age.

To the best of our knowledge, SkipBloom is the first algorithm for creating an appropriate synopsis of a blocking structure, while BlockSketch and SBlockSketch are the first methods for sufficiently summarizing a block for the needs of the matching phase of a record linkage task.

The rest of this paper is structured as follows: Section 2 presents the related work, while Section 3 outlines the building blocks utilized by our algorithms and provides the formal problem definition. Sections 4, 5, and 6 present our proposed algorithms from both a practical and a theoretical point of view. The results of our experimental evaluation, including a detailed comparison with baseline methods, are reported in Section 7, while Section 8 concludes this work.

2 RELATED WORK

A significant body of research work has been conducted in record linkage during the last four decades. This work has been nicely summarized in a number of survey articles [4, 9, 30]. However, only a very limited amount of work has targeted the area of near real-time record linkage, such as [5, 7, 8, 12, 15].

In [5], Christen et al. present an approach that involves a pre-processing phase, where the authors compute the similarities between commonly blocked values, using a set of inverted indexes. The authors use the double metaphone [3] method to encode the string values, which are then inserted into the inverted indexes. This scheme is extended in [27], where a heuristic method is presented to index the most frequent values of data fields. This method, though, requires a-priori knowledge of the values in certain fields and is not well-suited for settings where highly accurate results are needed. Ramadan and Christen in [26] utilize a tree structure where a sorting order is maintained according to a chosen field(s). A query record scans not only the node that is inserted, but also its neighboring nodes where similar records may also reside. Nevertheless, the distance computations that should be performed may degrade considerably the performance of this method in online settings.

Dey et al. in [7] develop a matching tree to speed up the decision about the matching status for a pair of records, so that

it can be made without the need to compare all field values. However, the performance of this method depends heavily on the training of the matching tree, which requires a large number of record pairs. Moreover, the authors do not draw any attention to the acute problem of reducing the record pairs comparisons. Ioannou et al. in [8] resolve queries under data uncertainty, using a probabilistic database. The effectiveness of their method heavily depends on the potential of the underlying blocking mechanism, which is used implicitly, to produce blocks of high-quality. In [12], Altwajry et al. propose a set of semantics to avoid resolving certain record pairs. Their scheme, however, focuses on how to resolve generic selection queries (e.g., range queries), rather than on minimizing the query time.

There is also another body of related literature that deals with *progressive* record linkage (e.g., [10, 24, 31]). These techniques report a large number of matching pairs *early* during the linkage process and are quite useful in the event of an early termination of the linkage process, or when there is limited time available for the generation of the complete result set.

The solutions proposed by Whang et al. in [31] and Papenbrock et al. in [24] are empirical and rely heavily on lexicographically sorting the input records to formulate clusters of similar records. Although the sorting technique is quite effective in finding similar values in certain cases, it cannot guarantee identification of matching record pairs. Consider, for example, the similar strings ‘Jones’ and ‘Kones’, where the first letter has been mistyped; using [24, 31], the corresponding records would definitely reside in different clusters (assuming a large number of records). Consequently, the corresponding pair of records would never be considered as matching.

More recently, Firmani et al. [10] introduced two progressive strategies that provide formal guarantees of maximizing recall, focusing though only on minimizing the number of queries to an oracle (which is an entity that replies correctly about the linkage status of a pair) and not on minimizing the running time. Both strategies implicitly assume an underlying blocking mechanism that has been applied on the data sets, and heavily rely on the effectiveness of that blocking mechanism. Their most serious shortcoming is the excessive amount of time-consuming similarity computations, which need to be performed between the formulated pairs in the blocks, *without achieving any increase in recall*. For example, in a data set of 3 million records (including the query set), more than 1.3 *billion* similarity computations should be performed without reporting any results!

There is also another body of work, termed as *meta-blocking* [22, 23], which investigates how to restructure the generated blocks with the aim of discarding redundant comparisons. Meta-blocking techniques, however, conduct a cumbersome transformation of a blocking structure into a graph, which renders these techniques not applicable to online settings.

In Section 7, we elaborate further on the approaches of Christen et al. and Firmani et al., which are the state-of-the-art methods with which we compare our proposed techniques.

3 BACKGROUND AND PROBLEM STATEMENT

In this section, we first introduce the necessary background and terminology for the understanding of our proposed schemes, and subsequently derive the problem statement.

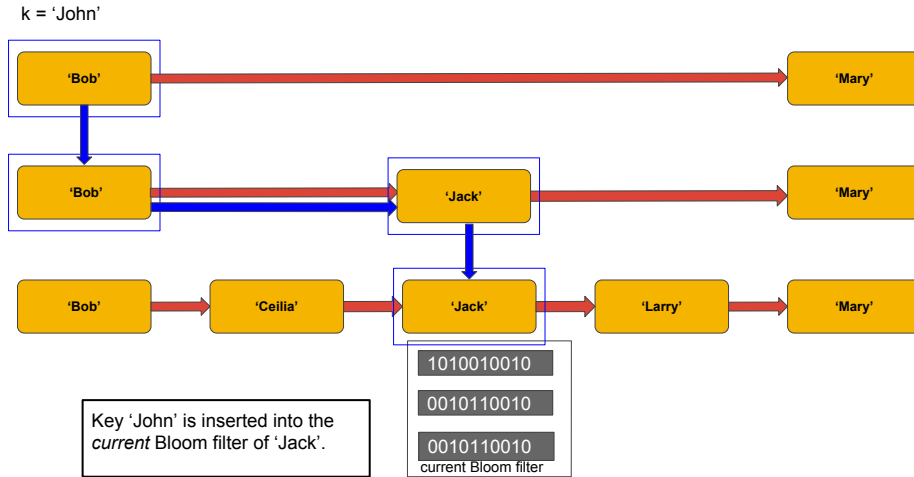


Figure 1: SkipBloom inserts and locates a key in logarithmic time using a small amount of main memory. The blue rectangles and arrows indicate the route to locate the nearest key to k .

3.1 Skip List

A skip list [25] is a probabilistic data structure that is designed to provide fast access to an ordered set of items. It is actually a sequence of lists, or *levels*, where the first list, termed as the *base level*, contains all the items inserted so far in sorted order. Each successive list is a copy of the previous with some elements skipped, until the empty list is reached. Its randomization lies in the number of levels an item will join, determined by tossing a fair coin¹. Each item of each list is linked to the same item in the previous list, as well as to the next item at the same level. The query operation for an item starts at the top-level, by horizontally scanning the items therein until it encounters either the target item or a larger item. In the case of a larger item, the same process is repeated at the lower level until the base level is reached. The running time to insert an item, as well as to report the existence of an item, is $O(\log(n))$, where n is the number of inserted items.

3.2 Bloom Filter

A Bloom filter [2] is a probabilistic data structure for representing a large number of items using a small number of bits, which are initialized to 0, to efficiently support membership queries. Each item is hashed by a set of universal hash functions that map it to certain positions, chosen randomly and uniformly, in the Bloom filter. Accordingly, these positions are set from 0 to 1. Upon querying for an item, the same process is followed, where:

- one can definitely infer that this item has not appeared, if all retrieved positions are set to 0.
- one can conjecture that this item has appeared with certain probability, if all retrieved positions are set to 1. The probabilistic nature of the reply is due to the fact that these positions may have been set to 1 by other items and not the query item.

3.3 Problem Formulation

Consider two data custodians who own data sets A and B , respectively. For each record r of A (or B), the data custodians use

¹As long as *tails* come up, we add the item to each successive list. We terminate this process when we encounter *heads*.

a function $k = \text{block}(r)$ that generates the blocking key k of r . This key is used to locate a *target block* in the blocking structure to either insert r into the target block (blocking), or iterate all the records already found therein and compare them with r (matching). We use D_A and D_B to denote the set of blocking keys of each of these data sets. Moreover, we refer to the fraction $\mathcal{D} = \frac{|D_A \cap D_B|}{|D_B|}$, as the *overlap coefficient* between A and B .

In this work we introduce three algorithms, namely SkipBloom, BlockSketch, and SBLOCKSketch, for addressing the following problems²:

Problem Statement 1. Calculate the overlap coefficient for A and B , by accurately summarizing D_A and D_B using **sublinear memory requirements and sublinear running time** in the number of inserted blocking keys.

Problem Statement 2. For each query record of A (or, equivalently, B), find the set of its matching records from B (or, equivalently, A) in **constant running time**.

Problem Statement 3. For each query record of A (or, equivalently, B), find the set of its matching records from B (or, equivalently, A) in **constant time**, using also a **constant amount of main memory**.

4 THE OPERATION OF SKIPBLOOM

SkipBloom is an efficient blocking data structure that reports membership queries of blocking keys (derived from a large data set) to the blocking structure, using only a small footprint of main memory. It implements the following generic operations:

- $\text{query}(k)$: Reports the membership (*true* or *false*) of key k to SkipBloom.
- $\text{insert}(k)$: Inserts key k into SkipBloom.

The operation of SkipBloom is based on a skip list that implements a mechanism to locate efficiently a blocking key, as well

²SkipBloom aims to address Problem 1, BlockSketch targets Problem 2, while SBLOCKSketch tackles Problem 3.

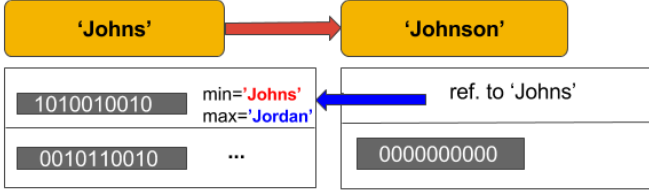


Figure 2: SkipBloom inserts a reference from the list of Bloom filters of ‘Johnson’ to the first Bloom filter of ‘Johns’, in order to maintain the consistency of the blocking mechanism.

as on a series of Bloom filters, which are used as fast memory-bounded buffers.

SkipBloom maintains, in expectation, \sqrt{n} blocks in main memory, stored in the base level of the skip list. Each such block, which is represented by its key³, includes a list of Bloom filters in order to store keys that have been driven by the mechanism of the skip list to this block. This actually means that the keys stored in the Bloom filters of a block are greater than the value of the corresponding key.

The operation of SkipBloom is illustrated in Figure 1. In this figure, a skip list is shown that contains five keys in the base level. Upon receiving a query record, which is first filtered by a blocking function to generate its key (e.g., $k = \text{‘John’}$), SkipBloom locates the block ‘Jack’ very fast, using the logarithmic runtime property of the underlying skip list. According to the operation of skip lists, this block is alphabetically the nearest key to k from the left. The next step is a simple insertion of k into a Bloom filter of ‘Jack’. Each block has an active Bloom filter, termed as *current*, and a number of inactive Bloom filters, which are used only during the query process, as we will shortly explain.

To answer a query on whether a certain key k exists or not, SkipBloom follows almost the same process as described above. Assume, for example, that SkipBloom receives the query $k = \text{‘Jonathan’}$. First, the skip list will be scanned to eventually locate ‘Larry’. Subsequently, each Bloom filter of this block will be iteratively queried until k is found, or the Bloom filters of ‘Larry’ are exhausted.

In what follows, we provide details that will justify certain design choices, such as the reason for maintaining a series of small (in length) Bloom filters in each block, instead of having a larger one. In order to populate the skip list with keys, we apply a simple Bernoulli random sampling algorithm that chooses each key with probability equal to $n^{-1/2}$. This sampling process ensures the uniform reflection of the distribution of keys from the data set to the skip list. This is an appealing feature, since SkipBloom easily tackles distribution anomalies, such as skews of certain ranges of keys, by choosing these keys and inserting them into the skip list to effectively reduce the bottleneck of certain keys and maintain uniformity (in expectation). Any uniform sampling method can be applied; we refer the interested readers to a comprehensive survey in [13].

If a large number of similar keys are generated, then the sampling routine will choose randomly similar keys to create the corresponding blocks. For example, consider the case of blocking a large number of surnames from the US census data. Then, possible blocks might be ‘Johns’, ‘Johnson’, and ‘Johnston’, which will

be created in this particular chronological order. Consequently, there will be keys other than ‘Johns’, e.g., ‘Jordan’ or ‘Jolly’, that will be inserted into the Bloom filters of ‘Johns’. These Bloom filters should be now transferred to (or referenced by) ‘Johnson’, and then to (by) ‘Johnston’. For this reason, we keep the number of keys that can be inserted into each Bloom filter small; this number will be accurately specified later. Moreover, we annotate each Bloom filter with its smallest and its greatest key, in terms of alphabetical order. By doing so, upon inserting ‘Johnson’, SkipBloom scans iteratively the Bloom filters of ‘Johns’ to locate Bloom filters that *might contain* ‘Johnson’, or any greater values. If such Bloom filters exist, a simple reference is established between the block of ‘Johnson’ and the corresponding Bloom filters. Figure 2 illustrates the reference of a block to a Bloom filter that belongs to the previous block.

Eventually, a record is stored into a key/value database system, maintaining its original blocking key, regardless of the block that was used in SkipBloom.

Algorithm 1 The query operation of SkipBloom.

Input: Skip list SL , query key k
Output: *true* if k is found, *false* otherwise
1: Key $p \leftarrow SL.query(k)$
2: **while** ($p.hasBloomfilters()$) **do**
3: $bf \leftarrow p.nextBloomfilter()$
4: **if** ($k \geq bf.min$ AND $k \leq bf.max$) **then**
5: **if** ($bf.member(k) == true$) **then**
6: **return true**
7: **end if**
8: **end if**
9: **end while**
10: **return false**

4.1 Algorithms

Algorithm 1 illustrates the query operation of SkipBloom. First, the skip list SL is queried to locate the nearest key p to the query k (line 1). Then, the Bloom filters that are both maintained and referenced by p ⁴ (line 2) are scanned iteratively to find k using the min and max values of each Bloom filter (line 4). If k is found, then the algorithm terminates (line 6). In case of composite keys, we perform a conjunction using the individual keys.

Algorithm 2 outlines the insertion of a key in SkipBloom. For each key k derived from each record, we determine with probability $\frac{1}{\sqrt{n}}$ whether k will be inserted into the skip list or not (line 1). In more detail, we generate a random value in $(0, 1)$ and then pick k if this value is less than $\frac{1}{\sqrt{n}}$. Since the generation of a random value is an expensive operation, we exploit the fact that the number of keys skipped between successive inclusions follow a geometric distribution [13]; accordingly, each time we pick a key, we generate the position of the next key, in the stream of records, that will be picked.

If a key k will be inserted into the skip list as a base level key, then a block is created after the nearest key to k (line 2). Then, SkipBloom has to locate each Bloom filter of p that may contain keys that should be now transferred to the newly created block of k (lines 4–8). In order to easily locate these Bloom filters, we annotate each Bloom filter used with the min and max keys it contains (line 5). The inclusion of a Bloom filter with a valid range of keys is achieved through a reference from p to k .

If a key will not be stored in the skip list, then the nearest key p to k is located in order to insert k in the current Bloom filter of

³Henceforth, *key* and *blocking key* will be used interchangeably.

⁴SkipBloom locates these Bloom filters performing a recursive process.

Algorithm 2 The insert operation of SkipBloom.

```

Input: Skip list  $SL$ , key  $k$ 
1: if ( $nextSample() == true$ ) then
2:    $Key\ p \leftarrow SL.insert(k)$   $\triangleright$  Key  $p$  is the nearest (previous) key to  $k$ 
3:    $List\ bfList \leftarrow k.createList()$   $\triangleright$  The list  $bfList$  that will
                                     host the Bloom filters of  $k$  is created
4:   for each  $bf$  in  $p$  do
5:     if ( $k \geq bf.min$  AND  $k \leq bf.max$ ) then
6:        $bfList.add(bf)$   $\triangleright$  A reference is added
                                     to each Bloom filter found in  $p$ 
                                     that might contain keys that belong to  $k$ 
7:     end if
8:   end for
9: else
10:   $Key\ p \leftarrow SL.query(q)$ 
11:   $bf \leftarrow p.getCurrentBloomFilter()$ 
12:   $bf.insert(k)$ 
13:  if ( $k \leq bf.min$ ) then
14:     $bf.min \leftarrow k$ 
15:  end if
16:  if ( $k \geq bf.max$ ) then
17:     $bf.max \leftarrow k$ 
18:  end if
19: end if
  
```

p (lines 10–12). Algorithm 2 eventually updates the min and max annotations of the current Bloom filter of p (lines 13–18).

4.2 Accuracy and Complexity Analysis

As we expect \sqrt{n} blocks in the base level of the skip list, where the sampling process ensures a uniform distribution of the corresponding blocking keys, the expected number c of keys in each block is:

$$E[c] = \frac{n}{\sqrt{n}} = \sqrt{n}. \quad (1)$$

By setting $u = \sqrt{n}/m$ to be the maximum number of keys that will be stored in each Bloom filter, where m is a constant value (e.g., $m = 10$), the number of Bloom filters in each block will be (in expectation) equal to m . Furthermore, the number m_{bt} of Bloom filters contained in block b at time t , specifies the upper and lower bound of the number n_{bt} of the distinct keys inserted, which is:

$$(m_{bt} - 1) \frac{\sqrt{n}}{m_{bt}} \leq n_{bt} \leq m_{bt} \frac{\sqrt{n}}{m_{bt}}. \quad (2)$$

The accuracy of SkipBloom to report the existence of a key depends on the false positive probability parameter fp of the Bloom filters. First, consider the event where a query key does not exist in any Bloom filter of the resulting block. The probability of reporting correctly this event, using one such Bloom filter, is $1 - fp$. Hence, the same probability by using collectively all the m Bloom filters is:

$$(1 - fp)^m, \quad (3)$$

since the content of a Bloom filter is independent from that of another Bloom filter.

In the case that a query key does exist in any⁵ Bloom filter of the resulting block, the probability of reporting this event is 1. Therefore, we bound from below the error probability of SkipBloom by $1 - (1 - fp)^m$.

Computational complexity: Based on Algorithm 1, the running time of querying SkipBloom is $O(\log(\sqrt{n}) + m + m\sqrt{n})$, where the first term denotes the time of scanning the skip list to locate the appropriate block, the second term denotes the time of scanning the Bloom filters found therein, and the last term is the time of scanning the Bloom filters referenced directly or indirectly by the chosen block.

⁵Since, we expect to have duplicate keys, it is quite natural that the same key may be stored into multiple Bloom filters of a block.

Algorithm 2 suggests that the running time of an insertion of a key into SkipBloom is $O(\log(\sqrt{n}) + m)$, where the two terms are the time of inserting a key into the skip list and the time of scanning the Bloom filters of the nearest key, respectively.

Memory complexity: The memory requirements of SkipBloom are $O(2\sqrt{n} + \sqrt{nm}) = O(\sqrt{n}(2 + m))$, because the skip list contains $O(2\sqrt{n})$ keys and each key in the base level of the skip list consists of $O(m)$ Bloom filters.

4.3 Using SkipBloom as a Synopsis of the Universe of Blocking Keys

SkipBloom can be used as a *synopsis*, termed also as *summarization*, of the universe of the blocking keys of a database, in order to facilitate an accurate pre-blocking process. During the execution of this process, the data custodians will resolve very fast the common blocks, which will be of great assistance in estimating the running time, in terms of the number of comparisons that will be needed (by exchanging the number of records in each common block). In turn, the data custodians will determine whether they will perform the linkage process or not, by considering several factors based on these preliminary results. For instance, if the number of common blocks is very small, then (a) the chances of identifying similar, or matching, record pairs are rather slim, and (b) the record linkage process itself may not be cost-effective.

Let us now consider the following scenario. Data custodian A generates a SkipBloom from database A , which is submitted to data custodian B . Subsequently, data custodian B iterates her blocking keys and queries the SkipBloom, which reports positive or negative answers for the existence of the query keys. This entails $O(n(\log(\sqrt{n}) + m + m\sqrt{n})) = O(n(\log(\sqrt{n}) + \sqrt{n}))$ running time,⁶ since each key of B is queried against the SkipBloom of A .

To further accelerate this process, data custodian B also generates a SkipBloom, to compile a uniform sample of keys and to use this SkipBloom to report membership queries. The keys found in the base level of the skip list are now queried against the SkipBloom of A , as illustrated in Figure 3.

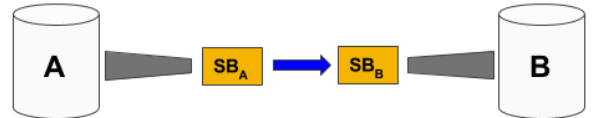


Figure 3: The blocking keys of the databases are packed into their corresponding synopses, each of which is implemented as a SkipBloom (symbolized by SB). These synopses are used to draw inferences about the source databases.

Since, the keys of B constitute a randomly and uniformly chosen sample, they can be used as input to a Monte Carlo simulation [21], which will estimate the proportion (or the number) of identical blocking keys between the data sets of the two data custodians. Using only the synopses, the data custodians will acquire a clear picture about the overlapping keys with certain approximation guarantees. Monte Carlo simulation requires $(\epsilon^2 \vartheta)^{-1}$ (ignoring a small constant factor) keys from B in order to exhibit relative error ϵ with high probability. Since the proportion of identical keys is unknown, we bound it from below with a reasonable value, e.g., $\vartheta = 0.05$, to approach the number \sqrt{n} of the sampled keys that

⁶We assume that the number of distinct blocking keys is n in both A and B .

'John', 'Jon'	
'John', 'Jon'	'John', 'Jonkers'
'John', 'Jones', 1970	'John', 'Jonker', 1975
'John', 'Jonas', 1985	'John', 'Jonkar', 1970

Figure 4: Illustration of a block with $\lambda = 2$ sub-blocks, whose key is $\langle \text{John}, \text{Jon} \rangle$. BlockSketch inserts records into the sub-blocks based on the distance of the key values of these records from the chosen representative(s). The sub-block for which one of its representatives exhibits the smallest distance from the key values of a record, is chosen as the target sub-block.

are contained in the SkipBloom of B . Even for a relatively small $n = 10^8$, the Monte Carlo simulation will provide its guarantees, since \sqrt{n} is greater than the required number of sampled keys for $\epsilon \geq 0.05$. The fraction of the overlapping keys found in the sample is used as an estimate for the overlap coefficient of the keys between the two databases. By comparing the synopses, we eventually achieve the much faster $O(\sqrt{n}(\log(\sqrt{n}) + \sqrt{n}))$ running time, compared to using only the synopsis of data custodian A .

5 THE OPERATION OF BLOCKSKETCH

The existence of blocks that contain a large number of records makes the *matching phase* (i.e., the comparison of query records against every record found in a target block) prohibitively expensive in highly demanding environments. The situation becomes even more challenging in environments where the matching record pairs have to be reported in near real-time.

To address this shortcoming, in this work we opt for a different strategy: we compare the query record with *a constant number of records of the target block*, which entails a bounded matching time. This optimization requires maintaining λ sub-blocks ($S_1, S_2, \dots, S_\lambda$) in each block, whose aim is to represent sufficiently the records inserted so far. In our proposed representation, a number of records play the key role of *representatives* for each sub-block. This allows to formulate groups of records inside each block that are more likely to match. We term our proposed algorithm as BlockSketch, because a small number of records comprise a sketch that represents sufficiently the records of an entire block. The concept of *sufficient representation* boils down to choosing representatives that exhibit certain distances from the corresponding blocking key. We note that BlockSketch can operate either autonomously or in conjunction with SkipBloom, where the latter will be used as a fast bounded memory to report whether a certain blocking key has appeared or not.

The fact that certain records are inserted into a block, using a blocking function, implies that all these records share some degree of similarity. Therefore, it is reasonable to assume that the distance between a key and a record⁷ will be upper bounded by $\lambda\theta$. Hence, BlockSketch formulates λ sub-blocks, each of which

⁷The distance either between a pair of records, or between a blocking key and a record, is determined by the distances of the certain field values, part of which usually make up the blocking key.

represents records with distances $\leq \theta, \leq 2\theta, \dots, \leq \lambda\theta$ from the key, where θ is the distance threshold of the keys of a pair of matching records. Upon receiving a key, BlockSketch aims to insert this record into the sub-block of the target block, where it is more likely to formulate matching record pairs. For this reason, each key is compared against all representatives found in a block, in order to locate the sub-block whose representative exhibits the smallest distance from the newly arrived key.

As an example, assume that we use edit distance as the similarity metric, $\theta = 2$ and $\lambda = 3$, and a blocking key is used that consists of the first three letters and the whole value of the *surname* and *given name* attributes, respectively. As Figure 4 shows, record $\langle \text{John}, \text{Jones}, 1970 \rangle$, whose key values exhibit a total distance of $2 \leq \theta$ from $\langle \text{John}, \text{Jon} \rangle$, is inserted into the 1-st sub-block, because of the representative $\langle \text{John}, \text{Jon} \rangle$. Similarly, $\langle \text{John}, \text{Jonker}, 1975 \rangle$, whose distance is $3 \leq 2\theta$ from $\langle \text{John}, \text{Jon} \rangle$, is inserted into the 2-nd sub-block, due to the comparison with the representative $\langle \text{John}, \text{Jonkers} \rangle$.

It is important to note that for threshold θ any metric that is used in record linkage processes can be supported, whether satisfying the triangle inequality or not. For example, a very commonly used metric in record linkage is the Jaro-Winkler similarity function [3], which takes on values in $[0, 1]$. Hence, one by setting the similarity threshold to θ' , and then by choosing $\theta = 1 - \theta'$, produces very reasonable sub-blocks.

The probability for a record to fall into a certain sub-block that holds its matching record, depends on the representatives of the target sub-block, as well as on the left and right neighboring sub-blocks. For instance, assume two neighboring sub-blocks with representatives *Jacks* and *Jackson*, respectively. The keys of these representatives comprise the values of the *surname* attribute. Key *Jackson* arrives, whose record is inserted into the identical sub-block of *Jackson*. At a later time, *Jacksn* arrives, that suffers from a typo, whose record is inserted into the sub-block of *Jacks*. We have thus missed the formulation of one matching record pair. BlockSketch tackles this deficiency by *using more than one representatives for each sub-block*⁸, so as to give more chances for grouping together matching record pairs. By doing so, if record a has been inserted into a sub-block, BlockSketch compares the key of its matching record b with more similar representatives to record a . To keep the number of representatives of a sub-block constant, whenever a key is chosen for inclusion in a sub-block, the algorithm tosses a coin to determine if this newly inserted key would be a representative as well. If it is chosen, a randomly picked old representative is evicted from the set of representatives.

As a last step, the query record is inserted into that sub-block which is maintained by a key/value database. The pairs formulated in this sub-block constitute the final result set.

5.1 Algorithm

Algorithm 3 outlines the basic operation of BlockSketch. For a query record q , the algorithm first retrieves an object S that contains the corresponding sub-blocks, either from a key/value database or from a cache structure in main memory (line 2). BlockSketch then iterates over the representatives of each sub-block and performs the distance computations between the key values of q and these representatives,⁹ whose results are stored in array u (line 5). The representative that exhibits the smallest

⁸The exact number of representatives will be specified later.

⁹A representative, being essentially a blocking key, has only key values.

distance from the key values of q specifies the sub-block (line 12) into which q is finally inserted (line 17). For ease of presentation, we omit from Algorithm 3 the details regarding the random choice and eviction of a representative from a sub-block.

Algorithm 3 The core operation of BlockSketch.

```

Input: Query record  $q$ 
1:  $k \leftarrow \text{block}(q)$ 
    ▶ Function  $\text{block}(\cdot)$  generates the blocking key, which will be used to look up the corresponding sub-blocks.
2:  $\text{SubBlocks } S \leftarrow \text{retrieve}(k)$ 
    ▶  $S$ , which is retrieved from secondary storage or from a cache structure, contains the sub-blocks of block  $k$ .
3: for  $i = 1$  to  $\lambda$  do
4:   for  $j = 1$  to  $\rho$  do
5:      $u[i][j] \leftarrow d(k, S[i][j])$ 
    ▶  $S[i][j]$  denotes the  $j$ -th representative of the  $i$ -th sub-block.
6:   end for
7: end for
8:  $\text{min} \leftarrow u[1][1]$ 
9: for  $i = 1$  to  $\lambda$  do
10:  for  $j = 1$  to  $\rho$  do
11:   if  $(\text{min} > u[i][j])$  then
12:     $\text{min} \leftarrow i$ 
    ▶ Find the  $i$ -th sub-block whose at least one of its representatives exhibits the smallest distance from  $k$ .
13:  end if
14: end for
15: end for
16:  $\text{represent}(k, \text{min})$ 
    ▶ Determine with a coin toss if  $k$  would be a representative for the chosen sub-block.
17:  $\text{insert}(q, k, \text{min})$ 
    ▶ Store  $q$  in a key/value database by setting the key as the concatenation of  $k$  and  $\text{min}$ .

```

5.2 Accuracy and Complexity Analysis

The probability of a record to fall into the correct sub-block is $1/\lambda$, since it completely relies on the distance from the corresponding representative. Hence, the inverse probability of a record not falling into the correct sub-block, and therefore not formulating a record pair, is $\leq 1 - 1/\lambda$. In order to amplify the probability of formulating a matching record pair, we give more chances for grouping together the two constituent records, by comparing each key with a number ρ of representatives, chosen randomly and uniformly from the underlying stream. We rigorously specify the required number of representatives that each sub-block should maintain, as the following lemma suggests.

LEMMA 5.1. *If a pair of records, which constitute a matching pair, has been brought in a certain block, then by maintaining $\rho = \lambda \ln(\frac{1}{\delta})$ representatives in each sub-block, this matching pair is detected with probability at least $1 - \delta$.*

PROOF. The probability of not detecting a matching pair that exists in a certain block is $(1 - \frac{1}{\lambda})^\rho$. We bound this probability above by δ and solve for ρ in the following:

$$(1 - \frac{1}{\lambda})^\rho < \delta \approx -\frac{\rho}{\lambda} < \ln(\delta) \iff \rho > \lambda \ln(\frac{1}{\delta}), \quad (4)$$

since $\ln(1 - \frac{1}{\lambda}) \leq -\frac{1}{\lambda}$. □

We subsequently apply the ceiling function on the value of ρ ($\lceil \cdot \rceil$), in order to select the smallest integer following ρ for the sake of optimality. ■

Computational complexity: The running time of BlockSketch is $O(\log n + \lambda\rho)$, which consists of the time

to retrieve a block from the database (which is logarithmic¹⁰), and the execution of the subsequent $\lambda \times \rho$ distance computations (ρ representatives for each of the λ sub-blocks).

Memory complexity: The storage requirements of BlockSketch are $O(\lambda n)$, where n is the number of blocking keys.

6 THE OPERATION OF SBLOCKSKETCH

Let us now suppose that the number of records, which are initiated from multiple sources, e.g., from different hospitals, is unbounded (or endless). This literally turns the record linkage scenario of a large number of records, into the record linkage of a stream of records. Therefore, BlockSketch will grow in both directions; it will not grow only in terms of sub-blocks, but also its number of blocks might unexpectedly grow considerably. Since our main memory is bounded, BlockSketch adapts its operation to record linkage tasks that involve streams of records.

In this version of BlockSketch, called SBBlockSketch, we bound the number of blocks, that are maintained in main memory, by an integer value μ which depends on the available main memory. Since the number of blocks is bounded, SBBlockSketch applies an eviction strategy, so as to insert a newly arrived blocking key from the stream, when there is not an empty slot to accommodate the corresponding block. We annotate each *live*¹¹ block with (a) the number of incoming records that generated its key, i.e., the number ξ of times this block has been chosen as the target block, and with (b) its age α , in terms of the number of times that this block has survived eviction, since its admission into main memory. We derive the eviction status of each block as follows:

$$es = e^{(w\xi - \alpha)}, \quad (5)$$

where factor w adjusts the weight of successes ξ of a block to its es . The intuition behind this scheme is that we promote (a) newer blocks against older ones, and (b) blocks that exhibit higher eligibility. The status of old blocks, that are additionally not chosen by the incoming records, will exponentially decay, which will result in their eviction from the main memory. SBBlockSketch is materialized by a hash table, which holds the live blocks, and the corresponding sub-blocks, and a priority queue, that is used to indicate which of these live blocks should be evicted in case of a newly arrived block (key).

Figure 5 illustrates the components of SBBlockSketch, namely the hash table T and the priority queue pq . T exists in main memory and contains a specified number μ of rows, each of which holds a block, as a function of the available main memory. Each row of T contains the sub-blocks of the corresponding block. The priority queue pq stores the eviction status of each live block in ascending order, so as to return the key of the block that holds the minimum eviction status. In the example shown in Figure 5, we observe that the block with key k_4 has survived $\alpha = 4$ evictions and has not been chosen as target block since its admission into T . These two events lead inevitably to its eviction, despite the existence of block k_2 , which has $\alpha = 10$ survivals, but it additionally exhibits $\xi = 6$ successes.

¹⁰For instance, LeveLDB (see <https://github.com/google/leveldb>) uses an in-memory highly efficient multi-level data structure, which enables logarithmic disk seeks in the number of stored blocking keys.

¹¹A *live* block is a block that is stored in main memory.

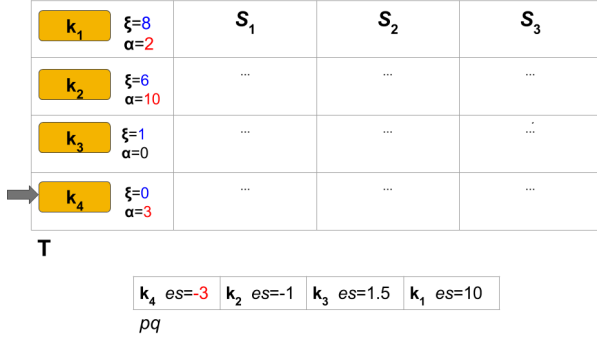


Figure 5: In this example, SBLOCKSKETCH uses a hash table T with $\mu = 4$ blocks, $\lambda = 3$ sub-blocks, and the weight of successes set to $w = 1.5$. On the arrival of an incoming new key, the block with key k_4 is evicted because of its low eviction status. The priority queue pq stores the eviction status (on a logarithmic scale) of each live block.

6.1 Algorithm

Algorithm 4 illustrates the operation of SBLOCKSKETCH, using a stream of data records. Upon receiving a record from the stream, the algorithm first derives its key, and then queries T (line 2). Only if this query is fruitless, SBLOCKSKETCH resorts to the structures of secondary storage (line 4). If the block that corresponds to the incoming record exists neither in T nor in secondary storage, then SBLOCKSKETCH initiates the eviction of the block from T that exhibits the minimum eviction status, as indicated by pq (line 7). Eventually, SBLOCKSKETCH computes the eviction status of each live block and rebuilds pq .

Algorithm 4 The eviction algorithm of SBLOCKSKETCH using a stream of records.

```

Input: Query record  $q$ 
1:  $k \leftarrow \text{block}(q)$ 
2:  $\text{SubBlocks } S \leftarrow T.\text{get}(k)$  ▷ Function  $\text{get}()$  retrieves an entry from hash table  $T$ .
3: if ( $S = \text{NULL}$ ) then
4:    $\text{SubBlocks } S \leftarrow \text{retrieve}(k)$ 
5: end if
6: if ( $S = \text{NULL}$ ) then
7:    $\text{SubBlocks } S \leftarrow pq.\text{poll}()$ ; ▷  $pq$  is a priority queue that holds the eviction status of each live block in ascending order.
▷ Function  $\text{evict}()$  transfers a certain block, which is essentially a structure of sub-blocks, from main memory into secondary storage.
8:    $S.\text{evict}()$ ; ▷ Function  $\text{calculateStatus}()$  computes the status of each live block and inserts it into  $pq$ .
9:    $\text{calculateStatus}()$ ;
10: end if

```

6.2 Accuracy and Complexity Analysis

The accuracy of SBLOCKSKETCH is not affected by the use of T , since the block in question might exist either in main memory or in secondary storage. However, T , whose operations are of $O(1)$ time, affects both running time and space.

Computational complexity: The running time depends on two mutually exclusive possibilities. The first one is when a block exists in T , where the running time is $O(\lambda)$ (see Section 5.2), while the other possibility is when a block should be evicted from T . The eviction requires accessing the priority queue, which is of $O(\sqrt{\mu})$ time, and then transferring the incoming block into T . The

Table 1: Technical characteristics of the data sets used. The blocking fields used, and their length (in characters) are shown in bold ($m = 5$).

	DBLP	NCVR	LAB
$ Q $	300K	500K	100K
$ A $	300M	500M	100M
fields	'author' [50%], 'venue' , 'year'	'given name' , 'surname' [50%], 'address' , 'town'	'assay' [6], 'result' 'year'
	$u = 3,465$	$u = 4,473$	$u = 2,000$

latter step consumes, as we discussed in Section 5.2, $O(\log(n))$ time in the number n of available blocks found in the secondary storage. Finally, we have to add the time to build the priority queue, which is $O(\mu \log(\sqrt{\mu}))$. Hence, the total running time for replacing a block is $O(\sqrt{\mu} + \log(n) + \mu \log(\sqrt{\mu}))$.

Memory complexity: The space occupied in main memory is exactly $O(\mu\lambda)$, where μ corresponds to the rows and λ to the cells of T (by assuming T as a two-dimensional array).

7 EXPERIMENTAL EVALUATION

For the experimental evaluation, we used three real-world data sets, namely (a) DBLP¹², which includes bibliographic data records, (b) NCVR¹³, which comprises a registry of voters, and (c) LAB¹⁴, which includes biological assays (e.g., albumin, hepatitis, or creatinine) and their corresponding results. The technical characteristics of these data sets are summarized in Table 1. For each record of each data set, denoted by Q , we generated 1,000 perturbed records, which were placed in a separate data set symbolized by A . We perturbed all the available fields using at most four edit, delete, insert, or transpose operations, chosen at random.

The blocking methods that were used for the needs of the evaluation were standard [4] and LSH blocking [18], which relies on the Locality-Sensitive-Hashing [11] technique. LSH blocking generates from a single record a certain number of blocking keys that are placed in multiple hash tables. This number of blocking keys is a function of several parameters [19] of LSH blocking, such as the distance threshold. The LSH technique is commonly used in the domain of record linkage [17, 18, 20, 29] because of its efficiency and accuracy guarantees. We used Hamming LSH blocking [18], in which records are embedded into the Hamming space using record-level Bloom filters [28]. LSH blocking implements redundant blocking, because a record is inserted into multiple independent blocks, which are accommodated into independent hash tables. In contrast, standard blocking inserts records that exhibit identical values, in an appropriately chosen blocking field(s), into the same block.

For performing the standard and the LSH blocking, we utilized LevelDB¹⁵ and LSHDB [16], respectively. The length of each Bloom filter, utilized by SkipBloom, was set to 32,000 bits for storing 5,000 keys, with false positive probability set to $fp = 0.05$.

We evaluated our schemes and their competitors according to the time needed, and the memory that was consumed to perform the record linkage process, as well as the recall and precision rates

¹²<http://dblp.uni-trier.de/xml>

¹³<http://dl.ncsbe.gov/index.html?prefix=data/>

¹⁴<https://dash-data.ucsd.edu/community/43>

¹⁵<https://github.com/google/leveldb>

that were achieved. We ran each experiment 20 times and plotted the average values in the figures. The software components were developed using the Java programming language (ver. 1.8) and the experiments were conducted in a virtual machine utilizing 4 cores of a Xeon CPU and 32GB of main memory.

7.1 Baseline Methods

We compared our schemes with two state-of-the-art methods for online record linkage. The first method, termed as INV [5], uses inverted indexes as its basic blocking structure. The main idea behind this method is the pre-computation of similarities between field values that have been inserted into the same block. An inverted index is used for this purpose, which stores the blocking keys encoded by the double metaphone method¹⁶. A weakness of this structure regards the storage of all field values of a record into the same set of indexes. As a result, one cannot be certain for a value encountered therein, to which field this value belongs. This ambiguity affects negatively the recall rates of INV.

The second method we compared against is the Edge Ordering strategy, termed as EO, which was introduced in [10]. EO utilizes an oracle, which is aware of the ground-truth, to resolve the matching status of a record pair. A graph is constructed by assuming each record pair, which materializes an edge connecting two vertices/records, formulated in each block. The algorithm performs all similarity computations in the target block in order to assign a probability estimate to each edge (pair) based on its similarity. In turn, EO selects those edges that are expected to maximize the recall, and submits them to the oracle that returns their matching status.

Both EO and INV utilize only key/value pairs, materialized by hash tables that map a key to list of record Id 's. These methods do not offer any component to report efficiently the membership of a certain key, or to adequately summarize the data set. Thus, in order to be fair in our comparison with these methods, we maintained the key/ Id 's mappings, as well as the entire records, in secondary storage.

Both the baseline methods and our proposed schemes used the Jaro-Winkler [3] function as the similarity measure, where the corresponding threshold was set to $\theta = 0.75$.

7.2 Experimental Results

In our first set of experiments, we evaluated the running time and memory performance, as well as the ability of the SkipBloom algorithm to provide accurate estimates in the pre-processing step of record linkage.

Figure 6a shows the total time needed to build the SkipBloom, by scaling the number of the streaming records using the NCVR data set. It is quite obvious that the time increases by a constant factor, depending on the number of records that are processed. The consumption of main memory is illustrated in Figure 6b, where SkipBloom exhibits almost linear performance. Specifically, although the number of records increases by 10 and 50 times, SkipBloom utilizes 0.6GB, 0.8GB, and 1.4GB of main memory, respectively. In contrast, a map data structure, symbolized by MAP, e.g., a HashMap in the Java programming language, exhibits a steep linear performance. In both scenarios, MAP throws fatal errors and terminates when it reaches the processing of 500M records.

¹⁶Using the double metaphone encoding method, 'SMITH' and 'SMYTH' are both encoded as 'SMO'.

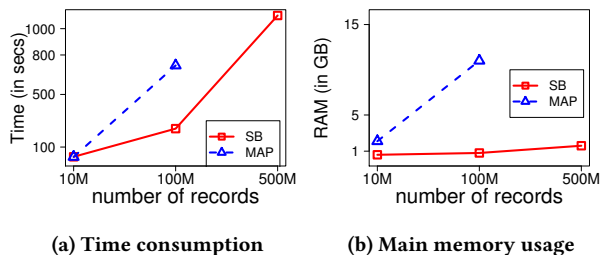


Figure 6: Scaling the number of records to measure the time and space requirements of SkipBloom.

Table 2: Time (in seconds) consumed by SkipBloom for reporting the existence of a key.

	10M	100M	500M
Time	0.000277	0.000315	0.000365

Table 3: Evaluating the accuracy of SkipBloom in estimating the fraction of matching pairs.

ϵ	DBLP	NCVR	LAB
.10	0.94 \pm .023	0.95 \pm 0.021	0.94 \pm 0.022
.05	0.97 \pm .022	0.98 \pm 0.021	0.98 \pm 0.024

Table 2 illustrates the time consumed by SkipBloom to report the existence of a key. We remind to the reader the probabilistic nature of SkipBloom, whose performance depends on the number of comparisons that will take place until the target block is located (which is $O(\log(\sqrt{n}))$). For this reason, we observe that SkipBloom almost consumes the same amount of time when it has to process either 100M or 500M records.

The accuracy of SkipBloom is evaluated by the fraction of overlapping keys it estimates using the above-mentioned data sets. Table 3 clearly shows that SkipBloom approximates the overlap coefficient of A and Q for each data set, where in the worst case it exhibits an error nearly equal to 0.06 (which is within its approximation guarantees specified by ϵ).

In the next set of experiments, we compared our schemes against EO and INV. Figures 7a and 7b display the recall rates achieved by all methods using standard and LSH blocking, respectively. We observe in Figure 7a that EO exhibits slightly better recall rates than BlockSketch, by using all data sets, although the differences lie in the small range [0.01, 0.04]. Also, INV falls short in formulating those matching pairs that exhibit a high degree of perturbation, which is due to the weakness of the double metaphone scheme to group together such pairs into the same blocks. The recall rates of DBLP and NCVR are also higher than LAB, which is due to the longer (in characters) blocking keys, which render them more tolerant to the perturbation errors. BlockSketch achieves to maintain high recall rates, although we have to stress that the underlying blocking method drives the whole linkage process. As Figure 7b suggests, LSH blocking, which leverages redundancy, scores much better rates than standard blocking. Only BlockSketch and EO can use LSH blocking, because they essentially run on top of the blocking mechanism.

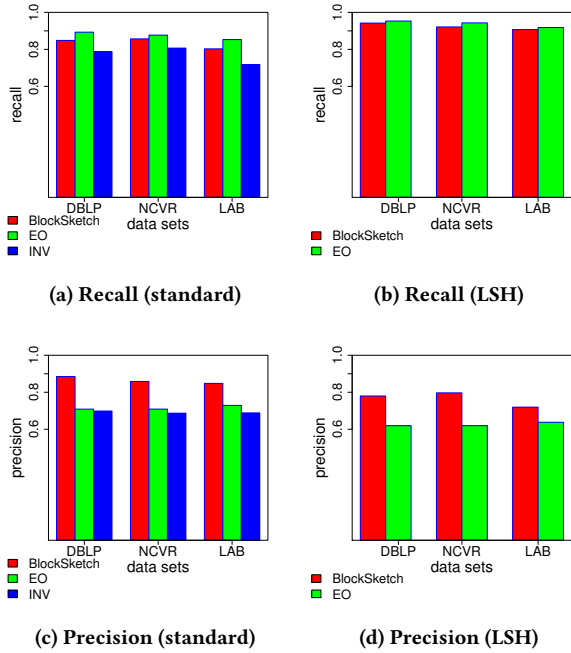


Figure 7: Measuring the recall and precision rates using standard blocking and LSH blocking.

On average, BlockSketch and EO achieve 10% and 8% higher recall rates, respectively, using LSH blocking.

Figures 7c and 7d show the precision rates using the two different blocking approaches described before. As one can observe, BlockSketch outperforms both EO and INV by a large margin, due to the effective categorization of records into the sub-blocks of each block. This minimizes significantly the required number of comparisons. Specifically, the precision rates of EO and INV fall by 18% and 21%, respectively, compared with the rates of BlockSketch. The reasons for this recession vary between the two methods. EO starts to produce meaningful recall rates after performing a large number of comparisons to derive the probability estimates for each pair. These comparisons, however, considerably reduce the precision rates. On the other hand, the double metaphone scheme of INV groups a large number of non-matching pairs into the same block, whose comparisons also result in low precision rates. The redundancy of LSH blocking accounts for the reduced precision rates of both BlockSketch and EO, as shown in Figure 7d, since both methods perform a larger number of comparisons for the pairs formulated in the blocks of each hash table. We observe though that BlockSketch retains its superiority over EO by scoring, on average, rates that are very close to 0.75. The time needed to perform the blocking step is illustrated in Figures 8a and 8b. EO and INV block each record a little faster than the combination of SkipBloom and BlockSketch, which for each insertion have to perform a constant number of comparisons with the representatives of the sub-blocks. Specifically, BlockSketch, through a single get operation, retrieves the representatives of a block from the database, as well as replaces them, through a single set operation, when needed. INV utilizes three hash tables to store the precomputed similarities, the encoded, and the original field values, which leads to certain delays.

Table 4: Average time (in seconds) for resolving a query record.

	DBLP	NCVR	LAB
standard	0.0051	0.0055	0.0045
LSH	0.0097	0.0098	0.0088

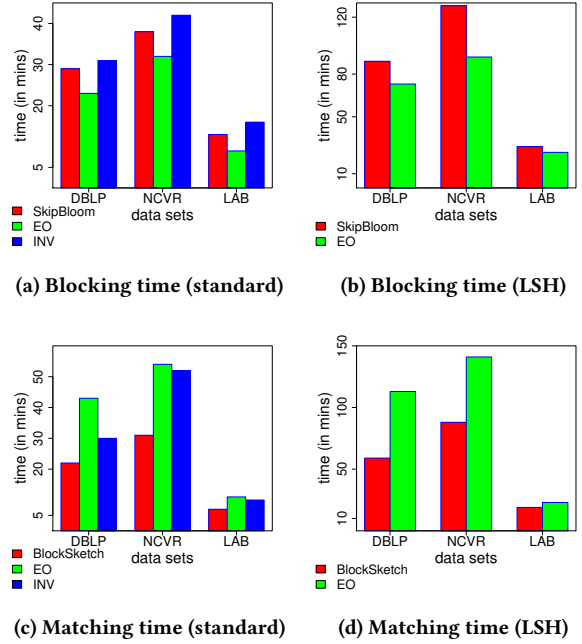
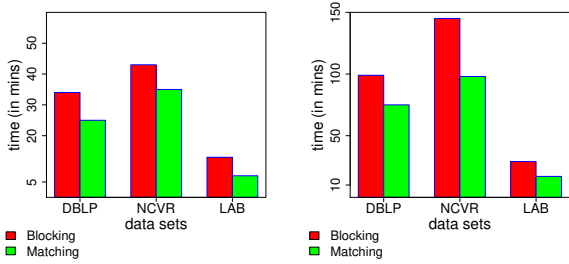


Figure 8: Measuring the time needed for blocking and matching for BlockSketch.

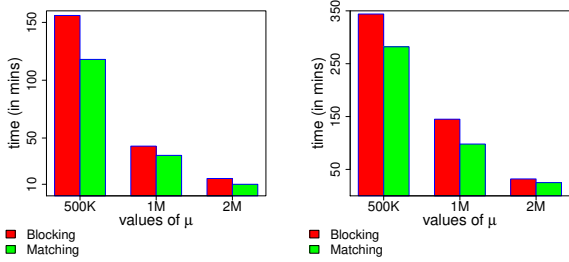
In Figures 8c and 8d, we present the time performance of BlockSketch and its competitors for resolving the query data sets, symbolized by Q (see Table 1), after having populated the blocking structures with the records of A . For each query record of Q , BlockSketch performs a constant number of comparisons in each target block, which results in superior performance. As Figure 8c suggests, BlockSketch is 2 \times and 1.5 \times faster than EO and INV, respectively, which both struggle to compare all records found in a block. Moreover, EO should build the graph to locate these record pairs that are expected to maximize the recall. Using LSH blocking, which is shown in Figure 8d, both BlockSketch and EO exhibit longer time rates, which are nearly 3 \times slower than before, due to the inherent redundancy of LSH. Since, a record pair might appear several times during the matching phase, for each record of Q , we utilize a map data structure¹⁷ to discard the comparisons of duplicate record pairs. Table 4 illustrates the time for resolving a single query record of Q during the matching phase. The constant number of distance computations for a single record accounts for the stable time performance of BlockSketch regardless of the size of the corresponding data set. In contrast, EO and INV consume running times which apart from the fact that in most cases they are almost the double of those of BlockSketch, they also depend on the number of records found in each block.

¹⁷The map structure is initialized for each record of Q .



(a) Running time (standard blocking) (b) Running time (LSH)

Figure 9: Measuring the time needed for blocking and matching for SBlockSketch.



(a) Running time (standard) (b) Running time (LSH)

Figure 10: Measuring the time needed for blocking and matching for SBlockSketch by varying μ using the NCVR data set.

In SBlockSketch, we initially set μ to a moderate size ($\mu = 1M^{18}$). In Figures 9a and 9b, we observe an average of 10% increase in time consumption than BlockSketch, only in NCVR and DBLP. The large number (over 60M) of distinct blocking keys that are generated in these data sets, resulted in relatively frequent evictions and disk seeks for the replacement of blocks in T . Nevertheless, the eviction status of highly selective (high ξ) but old (high α) blocks remained high during the blocking phase, which prevented their eviction from T . The running time of LAB remained almost intact due to the small number of blocking keys (about 10M) and the corresponding replacements. Since, SBlockSketch utilizes a single hash table T , LSH keys were formulated in a composite format *HashTableNo_Key* to accommodate all of them in T .

We next varied the values of μ and initiated the streaming of records of the NCVR data set. Figures 10a and 10b illustrate the time performance of SBlockSketch, where we observe that by doubling μ , we achieve significantly lower running time. For instance, by setting $\mu = 1M$, the corresponding time value is 43 minutes, which is almost 4 \times faster than the previous value (156 minutes) on the y-axis. In LSH blocking, the number of incoming records increases by a constant factor, which is the number of the LSH keys that are generated for each record. Since a large number of these keys are identical, the running time increases by 156% on average, as Figure 10b suggests, compared to the use of standard blocking.

¹⁸We had 32GB of main memory available.

Summary: Based on our conducted experiments, it becomes apparent that our proposed schemes are suitable for processing online queries for performing record linkage by using synopses of the blocking structures maintained in the persistent storage. They significantly outperform the state-of-the-art baselines, which rely their operation on memory-resident indexes regardless of the increasing volume of the underlying data sets.

8 CONCLUSIONS

In recent years, several applications have emerged which require access to consolidated information that has to be computed and presented in near real-time, through the linkage of records residing in voluminous disparate data sources. To address this need, we proposed the first summarization algorithms that operate in the blocking and matching steps of online record linkage to boost their performance. SkipBloom compiles a synopsis of the blocking structure of a data set using a small footprint of main memory, while BlockSketch compares each query record with a constant number of records in the target block, which results in a bounded matching time. Our experimental findings indicate that SkipBloom and BlockSketch outperform the state-of-the-art algorithms, in terms of the time needed, the memory used, and the recall and precision rates that are achieved during the linkage process. SBlockSketch utilizes a constant memory footprint to perform the linkage in settings that use streaming data.

REFERENCES

- [1] M. Bilenko, B. Kamath, and R. J. Mooney. 2006. Adaptive blocking: Learning to scale up record linkage. In *ICDM*. 87–96.
- [2] A. Broder and M. Mitzenmacher. 2002. Network Applications of Bloom filters: A Survey. In *Internet Mathematics*. 636–646.
- [3] P. Christen. 2012. *Data Matching - Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection*. Springer, Data-Centric Sys. and Appl.
- [4] P. Christen. 2012. A Survey of Indexing Techniques for Scalable Record Linkage and Deduplication. *TKDE* 12, 9 (2012), 1537–1555.
- [5] P. Christen, R. Gayler, and D. Hawking. 2009. Similarity-aware indexing for real-time entity resolution. In *CIKM*. 1565–1568.
- [6] W. W. Cohen and J. Richman. 2002. Learning to Match and Cluster Large High-Dimensional Data Sets for Data Integration. In *SIGKDD*. 475–480.
- [7] D. Dey, V. Mookerjee, and D. Liu. 2011. Efficient techniques for online Record Linkage. *TKDE* 23, 3 (2011), 373–387.
- [8] E. Ioannou, W. Nejdl, C. Niederee, and Y. Velegrakis. 2010. On-the-fly entity-aware query processing in the presence of linkage. *PVLDB* 3, 1 (2010), 429–438.
- [9] A. Elmagarmid, P. Ipeirotis, and V. Verykios. 2007. Duplicate Record Detection: A Survey. *TKDE* 19, 1 (2007), 1–16.
- [10] D. Firmani, B. Saha, and D. Srivastava. 2016. Online Entity Resolution Using an Oracle. In *PVLDB*, Vol. 9. 384–395.
- [11] A. Gionis, P. Indyk, and R. Motwani. 1999. Similarity Search in High Dimensions via Hashing. In *Vldb*. 518–529.
- [12] H. Altwaijry and D. Kalashnikov and S. Mehrotra. 2013. Query-driven Approach to Entity Resolution. In *PVLDB*, Vol. 6. 1846–1857.
- [13] P. J. Haas. 2016. Data-Stream Sampling: Basic Techniques and Results. *Data Stream Management: Processing High-Speed Data Streams* (2016), 13–44.
- [14] M.A. Hernandez and S.J. Stolfo. 1995. The Merge/Purge Problem for Large Databases. In *SIGMOD*. 127–138.
- [15] I. Bhattacharya and L. Getoor and L. Licamele. 2006. Query-time entity resolution. In *KDD*. 529–534.
- [16] D. Karapiperis, A. Gkoulalas-Divanis, and V. Verykios. 2016. LSHDB: a parallel and distributed engine for record linkage and similarity search. In *ICDM Demo*. 1–4.
- [17] D. Karapiperis, D. Vatsalan, V.S. Verykios, and P. Christen. 2016. Efficient Record Linkage Using a Compact Hamming Space. In *EDBT*. 209–220.
- [18] D. Karapiperis and V.S. Verykios. 2015. An LSH-based Blocking Approach with a Homomorphic Matching Technique for Privacy-Preserving Record Linkage. *TKDE* 27, 4 (2015), 909–921.
- [19] D. Karapiperis and V.S. Verykios. 2016. A fast and efficient Hamming LSH-based scheme for accurate linkage. *KAIS* (2016), 1–24.
- [20] H. Kim and D. Lee. 2010. Fast Iterative Hashed Record Linkage for Large-Scale Data Collections. In *EDBT*. 525–536.
- [21] R. Motwani and P. Raghavan. 1995. *Randomized Algorithms*. Cambridge Univ. Press.
- [22] G. Papadakis, G. Koutrika, T. Palpanas, and W. Nejdl. 2014. Meta-blocking: Taking Entity Resolution to the Next Level. *TKDE* 26, 8 (2014), 1946–1960.

- [23] G. Papadakis, G. Papastefanatos, and G. Koutrika. 2014. Supervised meta-blocking. In *PVLDB*. 1929–1940.
- [24] T. Papenbrock, A. Heise, and F. Naumann. 2015. Progressive Duplicate Detection. *TKDE* 27, 5 (2015), 1316 – 1329.
- [25] W. Pugh. 1990. Skip lists: a probabilistic alternative to balanced trees. *CACM* 33, 6 (1990), 668–676.
- [26] B. Ramadan and P. Christen. 2014. Forest-Based Dynamic Sorted Neighborhood Indexing for Real-Time Entity Resolution. In *CIKM*. 1787 – 1790.
- [27] B. Ramadan, P. Christen, H. Liang, R. Gayler, and D. Hawking. 2013. Dynamic Similarity-Aware Inverted Indexing for Real-Time Entity Resolution. In *PAKDD Workshops*. 47 – 58.
- [28] R. Schnell, T. Bachteler, and J. Reiher. 2009. Privacy-preserving Record Linkage using Bloom Filters. *Central Medical Inf. and Decision Making* 9 (2009).
- [29] R. Steorts, S. Ventura, M. Sadinle, and S. Fienberg. 2014. A Comparison of Blocking Methods for Record Linkage. In *PSD*. 253–268.
- [30] D. Vatsalan, P. Christen, and V.S. Verykios. 2013. A Taxonomy of Privacy-Preserving Record Linkage Techniques. *Inf. Sys.* 38, 6 (2013), 946 – 969.
- [31] S. E. Whang, D. Marmaros, and H. Garcia-Molina. 2013. Pay-as-you-go Entity Resolution. *TKDE* 25, 5 (2013), 1111–1124.
- [32] S. E. Whang, D. Menestrina, G. Koutrika, M. Theobald, and H. Garcia-Molina. 2009. Entity resolution with iterative blocking. In *SIGMOD*. 219–232.

Continuous Monitoring of Pareto Frontiers on Partially Ordered Attributes for Many Users

Afroza Sultana

University of Texas at Arlington
Arlington, Texas
afroza.sultana@mavs.uta.edu

Chengkai Li

University of Texas at Arlington
Arlington, Texas
cli@uta.edu

ABSTRACT

We study the problem of *continuous object dissemination*—given a large number of users and continuously arriving new objects, deliver an object to all users who prefer the object. Many real world applications analyze users’ preferences for effective object dissemination. For continuously arriving objects, timely finding users who prefer a new object is challenging. In this paper, we consider an append-only table of objects with multiple attributes and users’ preferences on individual attributes are modeled as *strict partial orders*. An object is preferred by a user if it belongs to the *Pareto frontier* with respect to the user’s partial orders. Users’ preferences can be similar. Exploiting shared computation across similar preferences of different users, we design algorithms to find *target users* of a new object. In order to find users of similar preferences, we study the novel problem of clustering users’ preferences that are represented as partial orders. We also present an approximate solution of the problem of finding target users which is more efficient than the exact one while ensuring sufficient accuracy. Furthermore, we extend the algorithms to operate under the semantics of sliding window. We present the results from comprehensive experiments for evaluating the efficiency and effectiveness of the proposed techniques.

1 INTRODUCTION

Many applications serve users better by disseminating objects to the users according to their preferences. User preferences can be modeled via a variety of means including *collaborative filtering* [19], *top-k ranking* [7, 8], *skyline* [2], and *general preference queries* [5, 12]. In various scenarios, users’ preferences stand or only change occasionally, while the objects keep coming continuously. Such scenarios warrant the need for a capability of continuous monitoring of preferred objects. While previous studies have made notable contributions on continuous evaluation of skyline [14, 28] and top-k queries [29], we note that two important considerations are missing from prior works:

- *Many users*: There may be a large number of users and the users may have similar preferences. Prior studies focus on the query needs of one user and thus their algorithmic solutions can only be applied separately on individual users. A solution can potentially attain significant query performance gain by leveraging users’ *common preferences*.
- *Partially ordered attributes*: Prior works focus on top-k and skyline queries. In multi-objective optimization, a more general concept than skyline is *Pareto frontier*. Consider a table of objects with a set of attributes. An object is *Pareto-optimal* (i.e., it belongs to the Pareto frontier) if and only if it is not dominated by any other object [1, 13]. Object y dominates x if and only

if y is better than or equal to x on every attribute and is better on at least one attribute. In defining the *better-than* relations, most studies on skyline queries assume a total order on the ordinal or numeric values of an attribute, except for [17, 30] which consider strict partial orders. The psychological nature of human’s preferences determines that it is not always natural to enforce a total order. Oftentimes real-world preferences can only be modeled as strict partial orders [5, 12, 17].

Consider the following motivating applications which monitor Pareto frontiers on partially ordered attributes for many users.

- *Social network content and news delivery*: It is often impossible and unnecessary for a user to keep up with the plethora of updates (e.g., news feeds in Facebook) from their social circles. When a new item is posted, if the item is Pareto-optimal with respect to a user, it can be displayed above other updates in the user’s view. Similar ideas can be adopted by mass media to ensure their news reaches the right audience. User preferences can be modeled on content creator, topic, location, and so on. Enforcing total orders on such attributes is both cumbersome and unnatural.
- *Publication alerts*: Bibliography servers such as PubMed and Google Scholar can notify users about newly published articles matching their preferences on venues and keywords. Such attributes do not welcome total orders either.
- *Product recommendation*: When a new product becomes available, a retailer can notify customers who may be interested. It can distill customers’ preferences on product specifications (e.g., brand, display and memory for laptops) from profiles, past transactions and website browsing logs. Example 1.1 discusses this application more concretely.

Example 1.1. Consider an inventory of laptops in Table 1 and customers’ preferences on individual product attributes (display, brand and CPU) modeled as strict partial orders in Table 2. For an attribute, the corresponding strict partial order is depicted as a directed acyclic graph (DAG), more specifically a Hasse diagram. Given two values x and y in the attribute’s domain, the existence of a path from x to y in the DAG implies that x is preferred to y . With respect to customer c_1 and attribute brand, the path from *Lenovo* to *Toshiba* implies that c_1 prefers *Lenovo* to *Toshiba*. There is no path between *Toshiba* and *Samsung*, which indicates c_1 is indifferent between the two brands.

The strict partial orders on various attributes together represent a customer’s preferences on objects. For instance, c_1 prefers $o_2 = \langle 14, \textit{Apple}, \textit{dual} \rangle$ to $o_1 = \langle 12, \textit{Apple}, \textit{single} \rangle$, since they prefer 13–15.9 to 10–12.9 on display and *dual* to *single* on CPU. With regard to o_1 and $o_3 = \langle 15, \textit{Samsung}, \textit{dual} \rangle$, c_1 does not prefer one over the other because, though they prefer 13–15.9 to 10–12.9 and *dual* to *single*, they prefer *Apple* to *Samsung* on brand.

According to the data in Tables 1 and 2, if the existing products are o_1 to o_{14} (ignore o_{15} and o_{16} for now), the Pareto frontiers of c_1 and c_2 are $\{o_2\}$ and $\{o_2, o_3\}$, respectively. Suppose $o_{15} = \langle 16.5, \textit{Lenovo}, \textit{quad} \rangle$ just becomes available. For c_1 , o_{15} does not belong

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

	display	brand	CPU
o_1	12	Apple	single
o_2	14	Apple	dual
o_3	15	Samsung	dual
o_4	19	Toshiba	dual
o_5	9	Samsung	quad
o_6	11.5	Sony	single
o_7	9.5	Lenovo	quad
o_8	12.5	Apple	dual
o_9	19.5	Sony	single
o_{10}	9.5	Lenovo	triple
o_{11}	9	Toshiba	triple
o_{12}	8.5	Samsung	triple
o_{13}	14.5	Sony	dual
o_{14}	17	Sony	single
o_{15}	16.5	Lenovo	quad
o_{16}	16	Toshiba	single

Table 1: Product table.

	display	brand	CPU
c_1	<pre> 13-15.9 10-12.9 / \ 16-18.9 19-up / \ 9.9-under </pre>	<pre> Apple Lenovo Sony / \ Toshiba Samsung </pre>	<pre> dual / \ triple quad </pre>
c_2	<pre> 13-15.9 10-12.9 16-18.9 19-up 9.9-under </pre>	<pre> Lenovo / \ Apple Samsung Toshiba Sony </pre>	<pre> quad triple dual single </pre>
U	<pre> 13-15.9 10-12.9 16-18.9 19-up 9.9-under </pre>	<pre> Apple Lenovo / \ / \ Toshiba Sony Samsung </pre>	<pre> dual triple quad \ / \ single </pre>
\hat{U}	<pre> 13-15.9 10-12.9 16-18.9 19-up 9.9-under </pre>	<pre> Apple Lenovo / \ / \ Sony Samsung Toshiba </pre>	<pre> dual quad \ / \ triple single </pre>

Table 2: User preferences. $U=\{c_1, c_2\}$.

to the Pareto frontier. It is dominated by o_2 , because c_1 prefers 14-inch display over 16.5-inch, *Apple* over *Lenovo*, and *dual*-core CPU over *quad*-core CPU. However, o_{15} is a Pareto-optimal object for c_2 since it is not dominated by any other object according to c_2 's preferences. It is thus recommended to c_2 , and the Pareto frontier of c_2 is updated to $\{o_2, o_3, o_{15}\}$. \triangle

This paper **formulates the problem of continuous monitoring of Pareto frontiers**: given a large number of users and continuously arriving new objects, for each newly arrived object, discover all users for whom the object is Pareto-optimal. Users' preferences are modeled as strict partial orders, one for each attribute domain of the objects.

It is key to devise an efficient approach to this problem. The value of a Pareto-optimal object diminishes quickly; the earlier it is found to be worth recommendation, the better. For instance, a status update in a social network keeps getting less relevant since the moment it is posted; a customer's need for a product may be fulfilled by a less preferred choice, if an even better option was not shown to the customer in time.

A simple, brute-force approach is to, given a newly arrived object, compute for every user if the object belongs to the Pareto

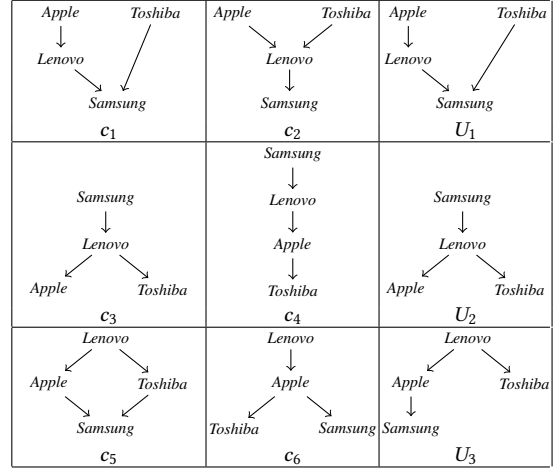


Table 3: User preferences with respect to brand. $U_1=\{c_1, c_2\}$, $U_2=\{c_3, c_4\}$, $U_3=\{c_5, c_6\}$.

frontier with respect to the user's preferences. This entails continuous maintenance of Pareto frontier for each and every user. The brute-force approach is subject to a clear drawback—repeated and wasteful maintenance of Pareto frontier for every user.

Sharing computation across users To tackle the aforementioned drawback, we partly resort to sharing computation across users. The challenge lies in the diversity of corresponding partial orders—a Pareto-optimal object with respect to one user may or may not be in the Pareto frontier for another user. Nonetheless, users have common preferences. In Table 2, both c_1 and c_2 prefer 13 – 15.9 inch display the most. Both prefer *Apple* and *Lenovo* to *Toshiba* and *Sony*, and they both prefer *single*-core CPU the least. In Table 2, U is a *virtual user* whose partial orders depict the common preferences of c_1 and c_2 . Intuitively, users having similar preferences can be clustered together.

We thus design algorithms to mitigate repetitive computation via sharing computation across similar preferences of users. To intuitively understand the idea, consider two example scenarios. i) If o is dominated by o' with respect to the common preferences of a set of users, then o is disqualified in Pareto-optimality for all users in the set. In Example 1.1, consider $o_{16}=(16, \textit{Toshiba}, \textit{single})$ as the new object. With respect to U , o_{16} is dominated by both $o_2=(14, \textit{Apple}, \textit{dual})$ and $o_{15}=(16.5, \textit{Lenovo}, \textit{quad})$. Therefore, o_{16} belongs to the Pareto frontier of neither c_1 nor c_2 . ii) Before the arrival of o_2 , obviously $o_1=(12, \textit{Apple}, \textit{single})$ is the only Pareto-optimal object for U , c_1 and c_2 . Now consider the entrance of o_2 . As o_1 is dominated by o_2 with respect to U , o_1 is replaced by o_2 in the Pareto frontier. This comparison is sufficient to decide that o_1 is dominated by o_2 for both c_1 and c_2 .

Clustering users To find users sharing similar preferences, we study the novel problem of clustering strict partial orders, which are used to model the preferences of both users and clusters. We measure the similarity between clusters and users by their common preferences. Such similarity measures factor in the different significance of preferences at various levels of the partial orders. Table 3 depicts six customers' preferences on brand, in which c_4 , c_5 , and c_6 prefer *Lenovo* to all other brands except that c_4 prefers *Samsung* over *Lenovo*. Consider the objects in Table 1. For both c_5 and c_6 , the Pareto frontiers contain $\{o_7, o_{10}, o_{15}\}$, while c_4 has $\{o_3, o_5, o_{12}\}$ as its Pareto frontier. We can say that c_5 and c_6 are more similar than c_4 and c_5 or c_4 and c_6 .

Approximation The clustering algorithm may produce clusters that comprise few users, due to diverse preferences. With

small clusters, the shared computation mentioned above may not pay off its overhead. Our response to this challenge is to use approximation. As in many data retrieval scenarios, insisting on exact answers is unnecessary and answers in close vicinity of the exact ones can be just good enough. Specifically, given a set of users, if a sizable subset of the users agree with a preference, the preference can be considered an approximate common preference. This relaxation eases the aforementioned concern regarding small clusters as more approximate common preferences lead to larger clusters. As an example, in Table 2, while c_2 does not share with c_1 the preference of *Apple* over *Samsung*, its preference does not oppose it either. We can consider “*Apple* over *Samsung*” as an *approximate common preference*. A possible set of approximate common preferences of c_1 and c_2 form the strict partial orders in the row for virtual user \hat{U} .

Alive objects Objects can have limited lifetime. The trends in social networks and news media change rapidly. Similarly, in any inventory, products become unavailable over time. In these scenarios users look for *alive* objects only. To meet this real-world requirement, we further extend our algorithms to operate under the semantics of a *sliding window* and thus to disseminate an object only during its lifespan.

In summary, the contributions of this paper are as follows:

- We study the problem of continuous object dissemination and formalize it as finding Pareto-optimal objects regarding partial orders. Given a large number of users and continuously arriving objects, our goal is to swiftly disseminate a newly arrived object to a user if the user’s preferences—modeled as strict partial orders on individual attributes—approve the object as Pareto-optimal.
- We devise efficient solutions exploiting shared computation across similar preferences of different users.
- We study the novel challenge of clustering user preferences represented as strict partial orders. Particularly we design similarity measures for such preferences.
- To address performance degradation due to small clusters, we present an approximate similarity measure that achieves high efficiency and accuracy of answers.
- We extend our proposed solutions to deal with Pareto frontier maintenance under sliding window.
- We conduct extensive experiments using simulations on two real datasets (a movie dataset and a publication dataset). The results demonstrate clear strengths of our solutions in comparison with baselines, in terms of execution time and efficacy.

2 RELATED WORK

Pareto-optimality is a subject of extensive investigation. Its study in the computing fields can be dated back to *admissible points* [1] and *maximal vectors* [13]. Börzsönyi et al. [2] introduced the concept of skyline—a special case of Pareto frontier—in which all attributes are numeric and amenable to total orders. Kießling [12] defined preferences as strict partial orders on which preference queries operate. After that, several studies specialized on skyline query evaluation over categorical attributes [3, 17, 18, 30], among which [17, 18, 30] particularly considered query answer maintenance and only [17, 30] allow partial orders on attribute values. Nevertheless, they all consider only one user and none utilizes shared computation across multiple users’ partial orders.

Given a set of objects, Wong et al. [25–27] identify the minimum set of preference relations that preclude an object from being in the Pareto frontier. This minimum set is the combination of each possible preference relation with regard to the values of all

unique objects in the set. In case of any update in the object set, the minimum disqualifying condition must be recomputed. Hence, it is not designed for continuously arriving objects.

Vlachou et al. [23, 24] and Yu et al. [29] aimed at finding all users who view a given object as one of their top- k favourites, i.e., the results of a reverse top- k query. Dellis et al. [6] studied reverse skyline query—selecting users to whom a given object is in the skyline. These works consider only numeric attributes. There is no clear way to extend them for categorical attributes or even partial orders.

All these studies, while about object dissemination, focused on different aspects of the problem than ours. Particularly, no previous studies on Pareto frontier maintenance have exploited shared computation across users’ preferences. Besides, as Sec. 5 shall explain, no prior work studied similarity measures for partial orders or how to cluster partial orders.

3 PROBLEM STATEMENT

Consider a set of users C and a table of objects O that are described by a set of attributes \mathcal{D} . For each user $c \in C$, their preference regarding O is represented by strict partial orders. For each attribute $d \in \mathcal{D}$, the strict partial order corresponding to c ’s preference on d is a binary relation over $dom(d)$ —the domain of d , as follows.

Definition 3.1 (Preference Relation and Tuple). Given a user $c \in C$ and an attribute $d \in \mathcal{D}$, the corresponding *preference relation* is denoted $>_c^d$. For two attribute values $x, y \in dom(d)$, if (x, y) belongs to $>_c^d$ (i.e., $(x, y) \in >_c^d$, also denoted $x >_c^d y$), it is called a *preference tuple*. It is interpreted as “user c prefers x to y on attribute d ”. A preference relation is irreflexive ($(x, x) \notin >_c^d$) and transitive ($(x, y) \in >_c^d \wedge (y, z) \in >_c^d \Rightarrow (x, z) \in >_c^d$), which together also imply asymmetry ($(x, y) \in >_c^d \Rightarrow (y, x) \notin >_c^d$). Δ

Definition 3.2 (Object Dominance). A user c ’s preferences regarding all attributes induce another strict partial order $>_c$ that represents c ’s preferences on objects. Given two objects $o, o' \in O$, c prefers o' to o if o' is identical or preferred to o on all attributes and o' is preferred to o on at least one attribute. More formally, $o' >_c o$ (called o' *dominates* o), if and only if $(\forall d \in \mathcal{D} : o.d = o'.d \vee o'.d >_c^d o.d) \wedge (\exists d \in \mathcal{D} : o'.d >_c^d o.d)$. If $(\forall d \in \mathcal{D} : o.d = o'.d)$, we say that o and o' are *identical*, denoted as $o = o'$. Δ

Definition 3.3 (Pareto Frontier). An object o is *Pareto-optimal* with respect to c , if no other object in O dominates it. The set of *Pareto-optimal objects* (i.e., the *Pareto frontier*) in O for c is denoted \mathcal{P}_c , i.e., $\mathcal{P}_c = \{o \in O \mid \nexists o' \in O \text{ s.t. } o' >_c o\}$. Note that the concept of skyline points [2] is a specialization of the more general Pareto frontier, in that the preference relations for skyline points are defined as total orders (with ties) instead of general strict partial orders. Δ

Definition 3.4 (Target Users). Given an object o , the set of all users for whom o belongs to their Pareto frontiers are called the *target users*. The target user set is denoted C_o , i.e., $C_o = \{c \in C \mid o \in \mathcal{P}_c\}$. Δ

Example 3.5. Consider Table 1 and Table 2. $O = \{o_1, o_2, \dots, o_{15}\}$ (ignore o_{16} for now), $C = \{c_1, c_2\}$, and $\mathcal{D} = \{\text{display, brand, CPU}\}$. With respect to c_1 , (10–12.9, 16–18.9), (*Apple, Samsung*) and (*dual, triple*) are some of the preference tuples on attributes display, brand and CPU, respectively. Similarly, for c_2 , (16–18.9, 19–up), (*Toshiba, Sony*) and (*triple, dual*) are some sample preference tuples.

$\mathcal{P}_{c_1} = \{o_2\}$, since all other objects are dominated by o_2 with respect to c_1 . $\mathcal{P}_{c_2} = \{o_2, o_3, o_{15}\}$, as o_2, o_3 and o_{15} dominate $\{o_1,$

$o_4, o_6, o_8, o_9, o_{13}$, $\{o_4, o_6, o_8, o_{13}\}$ and $\{o_4, o_5, o_7, o_{10}, o_{11}, o_{12}, o_{14}\}$, respectively. Therefore, $C_{o_2} = \{c_1, c_2\}$ and $C_{o_3} = C_{o_{15}} = \{c_2\}$. Objects other than o_2, o_3, o_{15} do not have target users in C , i.e., $C_o = \emptyset, \forall o \in O - \{o_2, o_3, o_{15}\}$. Δ

Problem Statement The problem of *continuous monitoring of Pareto frontiers* is, given a set of users C , their preference relations on attributes \mathcal{D} , and a set of continuously growing objects O with the latest object o , find C_o —the target users of o .

In this problem setting, we assume a sizable preference relation is available for each user. In reality, we have insufficient information about the preferences of a less active user, i.e., the corresponding partial orders may contain very few preference tuples. In the extreme case, a new user, for whom we have no information regarding their preferences, admits all objects as Pareto-optimal. Such less active users and new users are the subject of the well-known *cold-start* problem in recommendation systems, which is outside of the scope of this work.

4 SHARING COMPUTATION ACROSS USERS

Algorithm Baseline A simple method to our problem will check, for every user, whether a new object belongs to the corresponding Pareto frontier. The pseudo code of this approach, named Baseline, is shown in Alg. 1. Upon the arrival of a new object o , for every user c , it sequentially compares o with the current Pareto-optimal objects in \mathcal{P}_c . 1) If o is dominated by any o' or o is identical to o' , further comparison with the remaining objects in \mathcal{P}_c is skipped. In the case of o being dominated by o' , o is disqualified from being a Pareto-optimal object; if o is identical to o' , then o is Pareto-optimal, i.e., it is inserted into \mathcal{P}_c . 2) If o dominates any o' , o' is discarded from \mathcal{P}_c . It can be concluded already that o belongs to \mathcal{P}_c , but the comparisons should continue since o may dominate other existing objects in \mathcal{P}_c . 3) If o is not dominated by any object in \mathcal{P}_c , it becomes an element of \mathcal{P}_c . Readers familiar with the literature on skyline queries may have realized that the gist of the algorithm is essentially the basic skyline query algorithm [2]. The crux of its operation is based on an important property, that it suffices to compare new objects with only the Pareto-optimal objects, since any new object dominated by a non Pareto-optimal object must be dominated by some Pareto-optimal objects too.

Algorithm 1: Baseline

Input: C : all users; O : existing objects; o : a new object

Output: C_o : target users of o

```

1  $C_o \leftarrow \emptyset$ ;
2 foreach  $c \in C$  do
3    $\text{updateParetoFrontier}(c, o)$ ;
4 return  $C_o$ ;

Procedure:  $\text{updateParetoFrontier}(c, o)$ 
1  $\text{isPareto} \leftarrow \text{true}$ ;
2 foreach  $o' \in \mathcal{P}_c$  do
3   if  $o >_c o'$  then
4      $\mathcal{P}_c \leftarrow \mathcal{P}_c - \{o'\}; C_{o'} \leftarrow C_{o'} - \{c\}$ ;
5   else if  $o' >_c o$  then  $\text{isPareto} \leftarrow \text{false}; \text{break}$ ;
6   else if  $o'.\mathcal{D} = o.\mathcal{D}$  then  $\text{isPareto} \leftarrow \text{true}; \text{break}$ ;
7 if  $\text{isPareto}$  then
8    $\mathcal{P}_c \leftarrow \mathcal{P}_c \cup \{o\}; C_o \leftarrow C_o \cup \{c\}$ ;

```

With regard to a user c , the complexity of finding the Pareto frontier among n objects is $O(n^2)$. Alg. 1 needs $O(n^2 \cdot |C|)$ time to compute the Pareto frontiers for all users in C . The drawback of Baseline is it repeatedly applies the same procedure for every user. In terms of computation efficiency, the approach may become particularly unappealing when there are a large number of users and new objects constantly arrive. To counter this drawback, our idea is to share computations across the users that exhibit similar preferences. To this end, our method is simple and intuitive. If several users share a set of preference tuples, it is only necessary to compare two objects once, if they attain the attribute values in the preference tuples. If an object is dominated by another object according to these common preference tuples, it is dominated with respect to all users sharing the same preferences. This idea guarantees to filter out only “true negatives” for these users, and it only needs to further discern “false positives” for each individual user.

Definition 4.1 (Common Preference Tuple and Relation). Given a set of users $U \subseteq C$, an attribute $d \in \mathcal{D}$, and two values $x, y \in \text{dom}(d)$, if (x, y) belongs to preference relation $>_c^d$ for all $c \in U$, then it is called a *common preference tuple*. The set of common preference tuples of U on attribute d is denoted $>_U^d$, i.e., $>_U^d = \bigcap_{c \in U} >_c^d$. By definition, $>_U^d$ also represents a strict partial order (Theorem 4.2, proof omitted). We call it a *common preference relation*. It can be viewed as the preference of a virtual user that is denoted U . Δ

THEOREM 4.2. $>_U^d$ is a strict partial order. Δ

Since, for each $d, >_U^d$ is a strict partial order, the set of users’ preferences (i.e., the virtual user U ’s preferences) regarding all attributes in \mathcal{D} induce another strict partial order $>_U$ on objects.

Definition 4.3 (Pareto Frontier for U). An object o is *Pareto-optimal* with respect to U if no other object dominates it according to $>_U$. The Pareto frontier of O for U is denoted \mathcal{P}_U , i.e., $\mathcal{P}_U = \{o \in O \mid \nexists o' \in O \text{ s.t. } o' >_U o\}$. Δ

Example 4.4. From Table 2, $>_{c_1}^{\text{CPU}} = \{(dual, single), (dual, quad), (dual, triple), (triple, single), (quad, single)\}$ and $>_{c_2}^{\text{CPU}} = \{(dual, single), (triple, single), (quad, single), (triple, dual), (quad, dual), (quad, triple)\}$. According to Def. 4.1, the common preference relation of c_1 and c_2 is $>_{\{c_1, c_2\}}^{\text{CPU}} = \{(dual, single), (triple, single), (quad, single)\}$. Similarly we can derive $>_{\{c_1, c_2\}}^{\text{display}}$ and $>_{\{c_1, c_2\}}^{\text{brand}}$. In Table 2, the three partial orders are depicted in a row labeled as a virtual user U . The Pareto frontier of U is $\mathcal{P}_U = \{o_2, o_3, o_{10}, o_{15}\}$. Δ

THEOREM 4.5. Given any set of users U , for all $c \in U$, $\mathcal{P}_U \supseteq \mathcal{P}_c$ and $\overline{\mathcal{P}}_U \subseteq \bigcap_{c \in U} \overline{\mathcal{P}}_c$. Δ

Proof: We prove by contradiction. Suppose that there exists $c \in U$ such that $\mathcal{P}_U \not\supseteq \mathcal{P}_c$, which would mean there exists $o \in O$ such that $o \in \mathcal{P}_c$ and $o \notin \mathcal{P}_U$. That implies the existence of an $o' \in O$ such that $o' >_U o$ and $o' \not>_c o$. However, by Def. 4.1, $o' >_U o$ implies $o' >_c o$. Therefore, the existence of o' is impossible. This contradiction eventually leads to that $\mathcal{P}_U \supseteq \mathcal{P}_c$. Hence, $\mathcal{P}_U \supseteq \bigcup_{c \in U} \mathcal{P}_c$, which implies $\overline{\mathcal{P}}_U \subseteq \bigcap_{c \in U} \overline{\mathcal{P}}_c$ according to De Morgan’s laws.

LEMMA 4.6. Given any set of users U , for all $c \in U$, $\mathcal{P}_c = \{o \in \mathcal{P}_U \mid \nexists o' \in \mathcal{P}_U \text{ s.t. } o' >_c o\}$. Δ

Example 4.7. In Table 2, $\mathcal{P}_U = \{o_2, o_3, o_{10}, o_{15}\}$ and $\mathcal{P}_{c_1} \cup \mathcal{P}_{c_2} = \{o_2, o_3, o_{15}\}$. $\mathcal{P}_U \supseteq \mathcal{P}_{c_1} \cup \mathcal{P}_{c_2}$. Moreover, $\overline{\mathcal{P}}_U = \{o_1, o_4, o_5, o_6, o_7, o_8, o_9, o_{11}, o_{12}, o_{13}, o_{14}\}$ and $\overline{\mathcal{P}}_{c_1} \cap \overline{\mathcal{P}}_{c_2} = \{o_1, o_4, o_5, o_6, o_7, o_8, o_9, o_{10}, o_{11}, o_{12}, o_{13}, o_{14}, o_{15}\}$. $\overline{\mathcal{P}}_U \subseteq \overline{\mathcal{P}}_{c_1} \cap \overline{\mathcal{P}}_{c_2}$. Δ

Theorem 4.5 suggests an appealing quality of the common preference relations of U . By $\mathcal{P}_U \supseteq \mathcal{P}_c$, the Pareto frontier of U subsumes the Pareto frontier of every user member in U . What it means is that, if we simply compute the Pareto frontier of U , we get to retain all the objects that we eventually look for. Consider \mathcal{P}_c as the ground truth and \mathcal{P}_U as the predictions. The objects that are filtered out ($\overline{\mathcal{P}}_U$) are all “true negatives” and there are no “false negatives”. The set \mathcal{P}_U may contain “false positives”, which we just need to throw out after further verification, as Lemma 4.6 suggests.

This approach’s merit is the potential saving on object comparisons. For a cluster of users, many non Pareto-optimal objects may be filtered out altogether for all the users, without incurring the same comparisons repeatedly for each user.

To capitalize on the above ideas, our method must answer three questions. (1) How to find users sharing similar preferences? (2) For a set of similar users U , how to maintain the corresponding Pareto frontier \mathcal{P}_U based on their common preference relations $>_U^d$ for different attributes d ? (3) For each user c in U , how to discern the “false positives” in \mathcal{P}_U and thus find \mathcal{P}_c . Note that the second and the last challenges need to be addressed for constantly arriving new objects.

For (1), our method is to cluster users based on the similarity between their preference relations. While many clustering methods have been developed for various types of data, none is specialized in clustering partial orders. Our clustering method is discussed in Sec. 5. For (2) and (3), our algorithm takes a *filter-then-verify* approach and is thus named FilterThenVerify, of which the pseudo code is displayed in Alg 2.

Alg. FilterThenVerify Upon the arrival of a new object o , for every cluster U , FilterThenVerify compares o with the current members of \mathcal{P}_U based on the preference relations of the virtual user U . Various actions are taken, depending on the comparison outcomes, as follows:

I) If o dominates any o' in \mathcal{P}_U according to $>_U^d$ of all relevant d , o' is removed from \mathcal{P}_U (Line 7 of Procedure updateParetoFrontierU in Alg. 2). For every $c \in C$ such that $o' \in \mathcal{P}_c$, o' is also discarded from \mathcal{P}_c (Line 6 of Procedure updateParetoFrontierU).

II) If o is dominated by any o' in \mathcal{P}_U , then o does not occupy the Pareto frontier of any user in U (Theorem 4.5). Further operations involving o are unnecessary (Line 8 of Procedure updateParetoFrontierU).

III) After comparing o with all current objects in \mathcal{P}_U , if it is realized that o is not dominated by any o' , then o becomes a member of \mathcal{P}_U (Line 9 of updateParetoFrontierU). Furthermore, for each $c \in U$, o is further compared with the members of \mathcal{P}_c based on the preference relations of c , by using Procedure updateParetoFrontier of Alg.1 (Line 6 of Alg.2).

Example 4.8. In this example we explain the execution of FilterThenVerify on Table 1 and Table 2. Suppose users c_1 and c_2 form a cluster U , of which the preference relations are depicted in Table 2. The existing objects are o_1 to o_{14} , and $o_{15} = \langle 16.5'', \text{Lenovo, quad} \rangle$ is the object that just becomes available. Before o_{15} arrives, the Pareto frontier of U is $\mathcal{P}_U = \{o_2, o_3, o_7, o_{10}\}$. The algorithm starts by comparing o_{15} with each element in \mathcal{P}_U . As o_{15} dominates $o_7 = \langle 9.5'', \text{Lenovo, quad} \rangle$ according to U ’s

Algorithm 2: FilterThenVerify

Input: U_1, U_2, \dots, U_n : clusters of users; O : existing objects; o : a new object
Output: C_o : target users of o

```

1  $C_o \leftarrow \emptyset$ ;
2 foreach  $U \in \{U_1, U_2, \dots, U_n\}$  do
3    $isPareto \leftarrow \text{updateParetoFrontierU}(U, o)$ ;
4   if  $isPareto$  then
5     foreach  $c \in U$  do
6        $\text{updateParetoFrontier}(c, o)$ ; //Algorithm 1
7 return  $C_o$ ;

Procedure: updateParetoFrontierU ( $U, o$ )
1  $isPareto \leftarrow \text{true}$ ;
2 foreach  $o' \in \mathcal{P}_U$  do
3   if  $o >_U o'$  then
4     foreach  $c \in U$  do
5       if  $o' \in \mathcal{P}_c$  then
6          $\mathcal{P}_c \leftarrow \mathcal{P}_c - \{o'\}$ ;  $C_{o'} \leftarrow C_{o'} - \{c\}$ ;
7          $\mathcal{P}_U \leftarrow \mathcal{P}_U - \{o'\}$ ;
8       else if  $o' >_U o$  then  $isPareto \leftarrow \text{false}$ ; break ;
9 if  $isPareto$  then  $\mathcal{P}_U \leftarrow \mathcal{P}_U \cup \{o\}$  ;
10 return  $isPareto$ ;

```

preference relations, o_7 is discarded from \mathcal{P}_U . Before o_{15} arrives, o_7 belongs to \mathcal{P}_{c_2} and $C_{o_7} = \{c_2\}$. Hence, o_7 is removed from \mathcal{P}_{c_2} and C_{o_7} becomes empty. o_{15} does not dominate any other object in \mathcal{P}_U . It is not dominated by any either. Therefore, it is inserted into \mathcal{P}_U .

o_{15} is further compared with the existing members of \mathcal{P}_{c_1} and \mathcal{P}_{c_2} . It is dominated by $o_2 = \langle 14'', \text{Apple, dual} \rangle$ according to c_1 ’s preference relations. Thus it is not part of \mathcal{P}_{c_1} . According to c_2 ’s preferences, o_{15} does not dominate any existing Pareto optional object (except the aforementioned o_7 which by now is already discarded). Therefore \mathcal{P}_{c_2} is not further changed and o_{15} becomes part of \mathcal{P}_{c_2} . Overall, $C_{o_{15}} = \{c_2\}$.

Moreover, consider the arrival of $o_{16} = \langle 16'', \text{Toshiba, single} \rangle$ after o_{15} . In the process of comparing o_{16} with $\mathcal{P}_U = \{o_2, o_3, o_{10}, o_{15}\}$, it is realized that o_{16} is dominated by o_2 according to U ’s preference relations. Therefore, it does not belong to \mathcal{P}_U . It is thus unnecessary to further compare o_{16} with \mathcal{P}_{c_1} or \mathcal{P}_{c_2} . $C_{o_{16}} = \emptyset$. Thereby, updateParetoFrontierU acts as a sieve to filter out non Pareto-optimal objects such as o_{16} . In this way FilterThenVerify reduces computation cost by avoiding repeated comparisons with such objects. Δ

Complexity Analysis of Alg. 2 As we discussed earlier, given a user c , the complexity of finding Pareto frontier among the n objects is $O(n^2)$. Assume k is the number of clusters. With regard to the virtual user for each cluster U , the complexity of finding Pareto frontier \mathcal{P}_U among the n objects is $O(n^2 \cdot k)$ (calling Procedure updateParetoFrontierU in Line 3 of Alg. 2). Assume each \mathcal{P}_U on average has m objects. In Lines 4-6, Alg. 2 finds \mathcal{P}_c from \mathcal{P}_U for each user c in U (recall that $\mathcal{P}_U \supseteq \mathcal{P}_c$). As Lines 4-6 iterate for each cluster (Line 2), the algorithm eventually computes \mathcal{P}_c for each $c \in C$. Therefore, the complexity of finding Pareto frontier \mathcal{P}_c among the m objects is $O(m^2 \cdot |C|)$. Overall, FilterThenVerify needs $O(n^2 \cdot k + m^2 \cdot |C|)$ time to find the target users for all objects. We compare FilterThenVerify and Baseline in terms of time complexity. Apparently $k < |C|$ and $m < n$. Thus, $n^2 \cdot k < n^2 \cdot |C|$ and $m^2 \cdot |C| < n^2 \cdot |C|$.

5 SIMILARITY MEASURES FOR CLUSTERING USER PREFERENCES

This section discusses how to cluster users based on their preference relations. Our focus is on the similarity measures rather than the clustering method. The method we adopt is the conventional hierarchical agglomerative clustering algorithm [9]. At every iteration, the method merges the two most similar clusters. The common preference relation of the merged cluster U on each attribute d , i.e., \succ_U^d , is computed. It then calculates the similarity between U and each remaining cluster. Given two clusters U_1 and U_2 , their similarity $sim(U_1, U_2)$ is defined as the summation of the similarities between their preference relations on individual attributes, as follows. This resembles the high-level idea of using L_1 norm distance between centroids for measuring inter-cluster similarity in conventional hierarchical clustering.

$$sim(U_1, U_2) = \sum_{d \in \mathcal{D}} sim^d(U_1, U_2) \quad (1)$$

Individual users' and clusters' preference relations on attributes are strict partial orders. No prior work studied clustering approaches or similarity measures for partial orders. Similarity measures commonly used in clustering algorithms assume numeric or categorical attributes. Kamishima et al. [10, 11] and Ukkonen et al. [22] cluster total orders but not partial orders. Given two totally ordered attributes, these works use the comparative ranks of the corresponding values to measure similarity. Clearly, such similarity measures are not applicable for partially ordered attributes.

In this section we propose four different similarity functions for defining $sim^d(U_1, U_2)$.

1) Intersection size This is simply the size of the intersection of $\succ_{U_1}^d$ and $\succ_{U_2}^d$, i.e., the number of common preference tuples of all users in the two clusters U_1 and U_2 . It is defined as

$$sim_i^d(U_1, U_2) = |\succ_{U_1}^d \cap \succ_{U_2}^d| \quad (2)$$

Example 5.1. Table 3 shows three clusters U_1 ($\{c_1, c_2\}$), U_2 ($\{c_3, c_4\}$), and U_3 ($\{c_5, c_6\}$) and the common preference relation associated with each cluster on attribute *brand*. U_1 and U_2 do not share any preference tuple and thus $sim_i^{\text{brand}}(U_1, U_2) = 0$. U_1 and U_3 have (*Apple, Samsung*) and (*Lenovo, Samsung*) as common preference tuples, i.e., $sim_i^{\text{brand}}(U_1, U_3) = 2$. Similarly, U_2 and U_3 share (*Lenovo, Apple*) and (*Lenovo, Toshiba*), i.e., $sim_i^{\text{brand}}(U_2, U_3) = 2$. \triangle

2) Jaccard similarity The measure sim_i captures the absolute size of the intersection of two preference relations. It does not take into account their differences. Consider three clusters U_1, U_2 and U_3 such that $sim_i^d(U_1, U_2) = sim_i^d(U_1, U_3)$ (i.e., $|\succ_{U_1}^d \cap \succ_{U_2}^d| = |\succ_{U_1}^d \cap \succ_{U_3}^d|$) and $|\succ_{U_1}^d \cup \succ_{U_2}^d| < |\succ_{U_1}^d \cup \succ_{U_3}^d|$. We can argue that the similarity between U_1 and U_2 should be higher than (instead of equal to) that between U_1 and U_3 , because U_1 and U_2 have a larger percentage of common preference tuples than U_1 and U_3 . To address this limitation of sim_i , we define the *Jaccard similarity* between two preference relations as their intersection size over their union size, i.e., the ratio of common preference tuples to all preference tuples in the two preference relations. Formally,

$$sim_j^d(U_1, U_2) = \frac{|\succ_{U_1}^d \cap \succ_{U_2}^d|}{|\succ_{U_1}^d \cup \succ_{U_2}^d|} = \frac{sim_i^d(U_1, U_2)}{|\succ_{U_1}^d \cup \succ_{U_2}^d|} \quad (3)$$

Example 5.2. Continue Example 5.1. $\succ_{U_1}^{\text{brand}}$ and $\succ_{U_3}^{\text{brand}}$ have 6 preference tuples in total while $\succ_{U_2}^{\text{brand}}$ and $\succ_{U_3}^{\text{brand}}$ have 7. Thus, $sim_j^{\text{brand}}(U_1, U_3) = 2/6$ and $sim_j^{\text{brand}}(U_2, U_3) = 2/7$. \triangle

3) Weighted intersection size Intersection size and Jaccard similarity are based on the cardinalities of intersection and union sets of preference relations. In counting the cardinalities, they both treat all preference tuples equal. We argue that this is counter-intuitive. Values at the top of a partial order matter more than those at the bottom, in terms of their impact on which objects belong to the Pareto frontier. Accordingly we introduce *weighted intersection size*, a modified version of intersection size sim_i . In counting the common preference tuples of two preference relations, it assigns a weight to each preference tuple. Formally,

$$sim_{wi}^d(U_1, U_2) = \sum_{(v, v') \in \succ_{U_1}^d \cap \succ_{U_2}^d} \frac{1}{2} \times \left(\frac{1}{\min D(s, v) + 1} + \frac{1}{\min D(s, v') + 1} \right) \quad (4)$$

In the above equation, with regard to an attribute d , the similarity between two clusters' preference relations is a summation over their common preference tuples. For each common preference tuple (v, v') , it computes the average weight of the better value v with respect to U_1 and U_2 , respectively. Given a cluster U , S_U^d is the set of *maximal values* in the partial order \succ_U^d and $D(s, v)$ for each $s \in S_U^d$ is the shortest distance from s to v in \succ_U^d . The weight of v in U is the inverse of the minimal distance from any maximal value to v (plus 1, to avoid division by zero). The concept of maximal value is defined as follows.

Definition 5.3 (Maximal Value). With regard to \succ_U^d , value $x \in \text{dom}(d)$ is a *maximal value* if no other value in $\text{dom}(d)$ is preferred over x . The set of maximal values for \succ_U^d is denoted S_U^d . Formally, $S_U^d = \{x \in \text{dom}(d) \mid \nexists y \in \text{dom}(d) \text{ s.t. } (y, x) \in \succ_U^d\}$. \triangle

Example 5.4. Continue Example 5.1. The maximal values in $\succ_{U_1}^{\text{brand}}$, $\succ_{U_2}^{\text{brand}}$ and $\succ_{U_3}^{\text{brand}}$ are $S_{U_1}^{\text{brand}} = \{\text{Apple, Toshiba}\}$, $S_{U_2}^{\text{brand}} = \{\text{Samsung}\}$ and $S_{U_3}^{\text{brand}} = \{\text{Lenovo}\}$, respectively. In the partial order corresponding to $\succ_{U_1}^{\text{brand}}$, the minimal shortest distances to *Apple*, *Lenovo*, *Samsung*, and *Toshiba* from the maximal values $\{\text{Apple, Toshiba}\}$ are 0, 1, 1 and 0, respectively. The corresponding weights are 1, 1/2, 1/2 and 1. Similarly, in $\succ_{U_2}^{\text{brand}}$, the weights of *Apple*, *Lenovo*, *Samsung* and *Toshiba* are 1/3, 1/2, 1 and 1/3, respectively. In $\succ_{U_3}^{\text{brand}}$, the corresponding weights are 1/2, 1, 1/3 and 1/2, respectively.

U_1 and U_3 have (*Apple, Samsung*) and (*Lenovo, Samsung*) as common preference tuples. For the two better-values in these preference tuples—*Apple* and *Lenovo*, the average weights are both 3/4. The similarity $sim_{wi}^{\text{brand}}(U_1, U_3) = \frac{1+\frac{1}{2}}{2} + \frac{\frac{1}{2}+1}{2} = \frac{3}{2}$. Similarly, U_2 and U_3 have (*Lenovo, Apple*) and (*Lenovo, Toshiba*) as common preference tuples. In U_2 and U_3 , the average weight of *Lenovo*—the better-value in both common preference tuples—is 3/4. The similarity $sim_{wi}^{\text{brand}}(U_2, U_3) = \frac{\frac{1}{2}+1}{2} + \frac{\frac{1}{2}+1}{2} = \frac{3}{2}$. \triangle

4) Weighted Jaccard similarity This measure is a combination of the last two ideas—Jaccard similarity and weighted intersection size. As in Jaccard similarity, *weighted Jaccard similarity* computes the ratio of intersection size to union size. Similar to weighted intersection size, the values in a preference relation are assigned weights corresponding to their minimal shortest distances to the preference relation's maximal values. The measure's definition is as follows.

$$\begin{aligned}
& \text{sim}_{w_j}^d(U_1, U_2) \geq \sum_{(v, v') \in >_{U_1}^d \cap >_{U_2}^d} \frac{1}{2} \times \left(\frac{1}{\min_{s \in S_{U_1}^d} D(s, v) + 1} + \frac{1}{\min_{s \in S_{U_2}^d} D(s, v) + 1} \right) \\
& \quad \left| \sum_{(v, v') \in >_{U_1}^d \cup >_{U_2}^d} \frac{1}{2} \times \left(\frac{1}{\min_{s \in S_{U_1}^d} D(s, v) + 1} + \frac{1}{\min_{s \in S_{U_2}^d} D(s, v) + 1} \right) \right. \\
& = \text{sim}_{w_i}^d(U_1, U_2) \left| \left[\text{sim}_{w_i}^d(U_1, U_2) + \sum_{(v, v') \in >_{U_1}^d - >_{U_2}^d} \frac{1}{\min_{s \in S_{U_1}^d} D(s, v) + 1} \right. \right. \\
& \quad \left. \left. + \sum_{(v, v') \in >_{U_2}^d - >_{U_1}^d} \frac{1}{\min_{s \in S_{U_2}^d} D(s, v) + 1} \right] \right. \quad (5)
\end{aligned}$$

Example 5.5. Continue Example 5.4. Now $\text{sim}_{w_j}^{\text{brand}}(U_1, U_3) = \frac{\frac{3}{2}}{(1+1)+(1+1)+\frac{3}{2}} = \frac{3}{11}$, since $>_{U_1}^d - >_{U_3}^d = \{(Apple, Lenovo), (Toshiba, Samsung)\}$ and $>_{U_3}^d - >_{U_1}^d = \{(Lenovo, Apple), (Lenovo, Toshiba)\}$. Similarly, $\text{sim}_{w_j}^{\text{brand}}(U_2, U_3) = \frac{\frac{3}{2}}{(1+1+1)+(1+\frac{1}{2})+\frac{3}{2}} = \frac{3}{12}$, as $>_{U_2}^d - >_{U_3}^d = \{(Samsung, Lenovo), (Samsung, Apple), (Samsung, Toshiba)\}$ and $>_{U_3}^d - >_{U_2}^d = \{(Lenovo, Samsung), (Apple, Samsung)\}$. Note that $\text{sim}_{w_j}^{\text{brand}}(U_1, U_3) > \text{sim}_{w_j}^{\text{brand}}(U_2, U_3)$ although $\text{sim}_{w_i}^{\text{brand}}(U_1, U_3) = \text{sim}_{w_i}^{\text{brand}}(U_2, U_3)$. \triangle

6 APPROXIMATE USER PREFERENCES

Two conflicting factors have crucial impacts on the effectiveness of FilterThenVerify. One is the size of the common preference relations. The other is the size of the clusters. Specifically, the more preference tuples a cluster's users share, the more objects can be filtered out and thus the less verifications need to be done for individual users. On the contrary, the more users a cluster contains, the more repeated comparisons are avoided for these individual users. There is a clear tradeoff between these two factors, since larger clusters (i.e., more users in each cluster) naturally leads to smaller common preference relations.

Our approach to this challenge is *approximation*. As discussed in Sec. 1, it suffices for many applications to approximately identify target users. In this section, we show that we can find such approximation through a relaxed notion of common preference tuple, namely *approximate common preference tuple*. For a set of users, it allows a preference tuple to be absent from a tolerably small subset. If a sizable subset of the users agree with the preference tuple, it is considered an approximate common preference tuple. This relaxation addresses the aforementioned concern, since more approximate common preferences lead to larger clusters.

6.1 Approximate Common Preference Tuples and Relations

Based on the aforementioned objective, we procedurally construct approximate common preference relations. Before we provide its formal definition, we explain the intuition, as follows. Given a cluster of users, the resulting approximate common preference relation always includes the common preference tuples. The remaining possible preference tuples are considered in descending order of their frequencies, since preference tuples with higher frequencies are shared by more users. A preference tuple is included into the approximate common preference relation only if its reverse tuple is not included. This guarantees asymmetry. Furthermore, when a preference tuple is included, the transitive closure of the updated approximate common preference relation is also included.

This guarantees transitivity. Irreflexivity is guaranteed too since this procedure never considers preference tuples in the form of (x, x) . These altogether assure the constructed preference relation is a strict partial order. Given an append-only database of objects, a strict partial order ensures that preference query results are independent of the order by which objects are appended to the database. We denote the approximate common preference relation by $\widehat{\succ}_U^d$. It can be viewed as the preference of a virtual user (denoted \widehat{U}) on attribute d . Moreover, we denote the Pareto frontier of \widehat{O} for \widehat{U} as $\widehat{\mathcal{P}}_U$.

Definition 6.1 (Approximate Common Preference Tuple and Relation). Given a set of users $U \subseteq \mathcal{C}$, an attribute $d \in \mathcal{D}$ of which $|\text{dom}(d)| = m$, consider $A_1 \dots A_{P_2^m}$ which is an ordered permutation of all possible preference tuples $\{(x, y) \in \text{dom}(d) \times \text{dom}(d) \mid x \neq y\}$ such that $\text{freq}(A_i) \geq \text{freq}(A_{i+1})$ for $i \in [1, P_2^m - 1]$, in which $\text{freq}(A_i)$ denotes the percentage of users in U whose preference relations contain preference tuple A_i . The *approximate common preference relation* $\widehat{\succ}_U^d$ is defined as R_j in which j is the largest index $i \in [1, P_2^m]$ that satisfies the condition $(|R_i| < \theta_1 \wedge \text{freq}(A_i) > \theta_2) \vee \text{freq}(A_i) = 1$ where R_i is defined as

$$R_i = \begin{cases} \{A_i\} & \text{if } i = 1 \\ (R_{i-1} \cup \{A_i\})^+ & \text{if } R_{i-1} \cup \{A_i\} \text{ is a strict partial order} \\ R_{i-1} & \text{otherwise} \end{cases}$$

and θ_1 and θ_2 are two given thresholds. θ_1 limits the size of the resulting $\widehat{\succ}_U^d$ while θ_2 excludes infrequent preference tuples from $\widehat{\succ}_U^d$. \triangle

θ_1 and θ_2 regulate the size of $\widehat{\succ}_U^d$. A pair of large θ_1 and small θ_2 allows $\widehat{\succ}_U^d$ to include infrequent preference tuples. In such a case the approximate common preference relation becomes ineffective, since Procedure updateParetoFrontierU in Alg.2 may retain a large number of candidates that must be verified for each $c \in U$. On the other hand, a pair of small θ_1 and large θ_2 may limit $\widehat{\succ}_U^d$ to contain only $>_{U'}^d$, in which case the concern regarding small common preference relation remains.

As Def. 6.1 itself is procedural, it naturally corresponds to a greedy algorithm for constructing approximate preference relation $\widehat{\succ}_U^d$. The pseudo code GetApproxPreferenceTuples is in Alg. 3. First, all the common preference tuples are included (Lines 2-3). After that, preference tuples are considered in the order of frequency, as long as the two thresholds are satisfied (Line 4). For each preference tuple in consideration, if it together with all chosen tuples hitherto do not violate the properties of a strict partial order, their transitive closure is included into the approximate preference relation (Lines 6-7).

Example 6.2. We use Figure 1 to explain the execution of GetApproxPreferenceTuples. Figure 1a depicts three users' preference relations on brand. Suppose together these three users form a cluster. Assume $\theta_1 = 7$ and $\theta_2 = 60\%$.

Table 4 shows the frequencies of all possible preference tuples after sorting. For instance, since all users prefer *Apple* to *Toshiba*, the corresponding frequency is $3/3$; the frequency of *(Apple, Samsung)* is $2/3$ as two of these three users prefer *Apple* to *Samsung*. At first GetApproxPreferenceTuples includes the common preference tuple *(Apple, Toshiba)* into $\widehat{\succ}_U^d$. It then includes *(Apple, Samsung)*, *(Lenovo, Toshiba)*, and *(Toshiba, Samsung)* as approximate preference tuples too. Furthermore, upon the addition of *(Toshiba, Samsung)*, GetApproxPreferenceTuples includes *(Lenovo, Samsung)* as well since *(Lenovo, Toshiba)* and

Algorithm 3: GetApproxPreferenceTuples

Input: A_i : ordered permutation of all possible preference tuples, defined on $dom(d)$, in descending order of their frequencies among users U , θ_1 and θ_2 : thresholds

Output: $\widehat{\succ}_U^d$: approximate common preference relation of U on attribute d

```

1 for  $i = 1$  to  $P_2^{dom(d)}$  do
2   if  $freq(A_i) = 1$  then
3      $\widehat{\succ}_U^d \leftarrow \widehat{\succ}_U^d \cup \{A_i\}$ ; continue;
4   if  $|\widehat{\succ}_U^d| \geq \theta_1$  or  $freq(A_i) \leq \theta_2$  then
5     break;
6   if  $(\widehat{\succ}_U^d \cup \{A_i\})$  is a strict partial order then
7      $\widehat{\succ}_U^d \leftarrow (\widehat{\succ}_U^d \cup \{A_i\})^+$ ;
8 return  $\widehat{\succ}_U^d$ ;

```

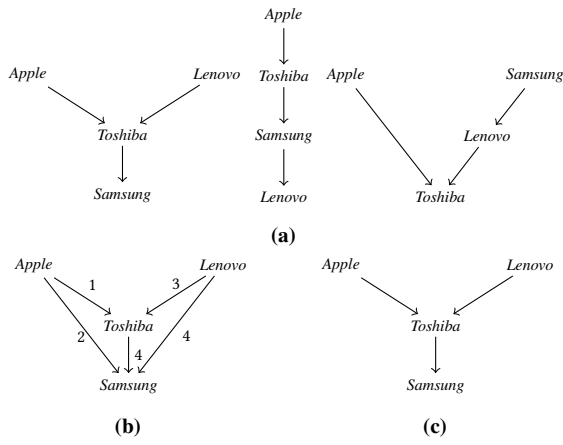


Figure 1: Execution of GetApproxPreferenceTuples. a) Input: the preferences of 3 users w.r.t. brand. b) The sequence of included approximate preference tuples. c) Output: the final Hasse diagram representation of the partial order.

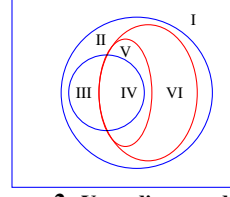
(A, T)	(A, S)	(L, T)	(T, S)	(S, L)	(A, L)	(L, S)	(T, L)	(S, T)	(L, A)	(T, A)	(S, A)
3/3	2/3	2/3	2/3	2/3	1/3	1/3	1/3	1/3	0/3	0/3	0/3

Table 4: All possible preference tuples in order of frequency. (A, L, S and T stand for Apple, Lenovo, Samsung and Toshiba.)

(Toshiba, Samsung) transitively induce it. The algorithm then considers (Samsung, Lenovo), which is disqualified since its reverse tuple (Lenovo, Samsung) is already included. Otherwise the tuples will not form a strict partial order. The algorithm stops at (Apple, Lenovo) because its frequency is below the threshold 60%. Fig. 1b illustrates the sequence of the included tuples and Fig. 1c depicts the output approximate preference relation in the form of a Hasse diagram. Δ

6.2 False Positives and False Negatives due to Approximation

FilterThenVerify (Alg.2) is extended to use approximate preference tuples and thus we rename it FilterThenVerifyApprox. The algorithm itself remains the same. Procedure updateParetoFrontierU maintains $\widehat{\mathcal{P}}_U$ as the candidate Pareto frontier. The algorithm eventually returns $\widehat{\mathcal{P}}_c$ for each user $c \in U$, in which $\widehat{\mathcal{P}}_c = \{o \in \widehat{\mathcal{P}}_U \mid \nexists o' \in \widehat{\mathcal{P}}_U \text{ s.t. } o' \succ_c o\}$, i.e., $\widehat{\mathcal{P}}_U \supseteq \widehat{\mathcal{P}}_c$. Thus, $\widehat{C}_o = \{c \in C \mid o \in \widehat{\mathcal{P}}_c\}$. We use the example below to explain its execution over approximate preference relations.



Set	Area Covered
\mathcal{O}	I,II,III,IV,V,VI
\mathcal{P}_U	II,III,IV,V,VI
$\widehat{\mathcal{P}}_U$	IV,V,VI
\mathcal{P}_c	III,IV
$\widehat{\mathcal{P}}_c$	IV,V

Figure 2: Venn diagram depicting \mathcal{O} , \mathcal{P}_U , $\widehat{\mathcal{P}}_U$, \mathcal{P}_c and $\widehat{\mathcal{P}}_c$. **Table 5: Areas covered by \mathcal{O} , \mathcal{P}_U , $\widehat{\mathcal{P}}_U$, \mathcal{P}_c and $\widehat{\mathcal{P}}_c$ in Fig.2.**

Approx.	Exact		
	Pareto frontier	Pareto frontier	Non Pareto frontier
	IV	IV	V
	Non Pareto frontier	III	I,II,VI

Table 6: Confusion matrix w.r.t. c.

Example 6.3. Reconsider Example 4.8, but use the approximate preference relations associated with virtual user \widehat{U} in Table 2. Upon the arrival of o_{15} , it is compared with the elements in $\widehat{\mathcal{P}}_U = \{o_2, o_7\}$. $\widehat{\mathcal{P}}_U$ becomes $\{o_2, o_{15}\}$ since o_{15} dominates o_7 . o_7 is then also removed from $\widehat{\mathcal{P}}_{c_2}$. o_{15} is further compared with $\widehat{\mathcal{P}}_{c_1} = \{o_2\}$ and $\widehat{\mathcal{P}}_{c_2} = \{o_2\}$, which does not lead to any further change. Overall, $\widehat{C}_{o_{15}} = \{c_2\}$. The target users using approximate preference relations remain identical to the exact ones, i.e., no loss of accuracy in this case. Δ

The rest of this section focuses on the accuracy of FilterThenVerifyApprox. It produces *false positives* if there exists such an o that $o \in \widehat{\mathcal{P}}_c$ but $o \notin \mathcal{P}_c$. It produces *false negatives* if there exists such an o that $o \notin \widehat{\mathcal{P}}_c$ but $o \in \mathcal{P}_c$. Below we present Theorems 6.5 and 6.7 to analyze how $\widehat{\mathcal{P}}_U$ and $\widehat{\mathcal{P}}_c$ relate to \mathcal{P}_U and \mathcal{P}_c .

LEMMA 6.4. Given a set of users U and an attribute d , the common preference relation \succ_U^d and an approximate common preference relation $\widehat{\succ}_U^d$ satisfy the following properties:

- 1) The approximate preference tuples are a superset of the common preference tuples, i.e., $\widehat{\succ}_U^d \supseteq \succ_U^d$.
- 2) If any preference tuple along with its reverse tuple do not belong to the approximate common preference relation, neither of them belongs to the common preference relation either, i.e., $(x, y) \notin \widehat{\succ}_U^d \wedge (y, x) \notin \widehat{\succ}_U^d \Rightarrow (x, y) \notin \succ_U^d \wedge (y, x) \notin \succ_U^d$. Δ

THEOREM 6.5. Given objects \mathcal{O} and users U , the Pareto frontier with regard to approximate common preference relations is a subset of the Pareto frontier with regard to common preference relations, i.e., $\widehat{\mathcal{P}}_U \subseteq \mathcal{P}_U$. Δ

Proof: We prove by contradiction. Suppose $\widehat{\mathcal{P}}_U \not\subseteq \mathcal{P}_U$, which would mean there exists $o \in \mathcal{O}$ such that $o \in \widehat{\mathcal{P}}_U$ and $o \notin \mathcal{P}_U$. That leads to the existence of an o' such that $o' \succ_U o$ and $o' \not\succeq_{\widehat{U}} o$. However, $o' \succ_U o$ implies $o' \succ_{\widehat{U}} o$ because $\widehat{\succ}_U^d \supseteq \succ_U^d$ for every d (Lemma 6.4). Therefore, the existence of o' is impossible. This contradiction proves that $\widehat{\mathcal{P}}_U \subseteq \mathcal{P}_U$. \blacksquare

LEMMA 6.6. Given any set of users U , for all user $c \in U$, $\widehat{\mathcal{P}}_U \supseteq \widehat{\mathcal{P}}_c$. Δ

THEOREM 6.7. Given any set of users U , for all user $c \in U$, $\widehat{\mathcal{P}}_U \cap \mathcal{P}_c \subseteq \widehat{\mathcal{P}}_c$. Δ

Proof: We prove by contradiction. Suppose $\widehat{\mathcal{P}}_U \cap \mathcal{P}_c \not\subseteq \widehat{\mathcal{P}}_c$, which would mean there exists $o \in \mathcal{O}$ such that $o \in \widehat{\mathcal{P}}_U \cap \mathcal{P}_c$ and $o \notin \widehat{\mathcal{P}}_c$. $o \notin \widehat{\mathcal{P}}_c$ implies the existence of an $o' \in \mathcal{O}$ such that $o' \in \widehat{\mathcal{P}}_c$ and $o' \succ_c o$ (since $o \in \widehat{\mathcal{P}}_U \cap \mathcal{P}_c$ and thus $o \in \widehat{\mathcal{P}}_U$ which

means $o' \not\prec_{\widehat{U}} o$. Since $o' \succ_c o$, $o \notin \mathcal{P}_c$ (Def. 3.3) and thus $o \notin \widehat{\mathcal{P}}_U \cap \mathcal{P}_c$. In other words, the existence of o' is impossible. This contradiction proves that $\widehat{\mathcal{P}}_U \cap \mathcal{P}_c \subseteq \widehat{\mathcal{P}}_c$. ■

Consider a cluster U and a user $c \in U$. The Venn diagram in Fig. 2 shows the effect of approximation through depicting \mathcal{O} (rectangle), \mathcal{P}_U (outer blue circle), $\widehat{\mathcal{P}}_U$ (outer red ellipse), \mathcal{P}_c (inner blue circle), and $\widehat{\mathcal{P}}_c$ (inner red ellipse). Besides, Table 5 elaborates the area covered by these sets while Table 6 shows the confusion matrix for c . Note that using approximate common preference relations results in false negatives (III). Mistakenly declaring III as not Pareto-optimal further allows false positives (V) to sneak in.

With these notations in place, we are ready to quantify the accuracy of FilterThenVerifyApprox using standard evaluation measures in information retrieval. Specifically, *precision* is the fraction of objects found by FilterThenVerifyApprox that are truly Pareto-optimal, i.e., $\frac{\sum_{c \in C} \widehat{\mathcal{P}}_c \cap \mathcal{P}_c}{\sum_{c \in C} \widehat{\mathcal{P}}_c}$. *Recall* is the fraction of Pareto-optimal objects that are correctly found by FilterThenVerifyApprox, i.e., $\frac{\sum_{c \in C} \widehat{\mathcal{P}}_c \cap \mathcal{P}_c}{\sum_{c \in C} \mathcal{P}_c}$. With regard to a specific user c , the algorithm's precision, recall and accuracy can be represented using the areas in Fig. 2, as follows.

$$precision = \frac{|IV|}{|IV \cup V|} \quad (6)$$

$$recall = \frac{|IV|}{|III \cup IV|} \quad (7)$$

$$accuracy = \frac{|I \cup II \cup IV \cup VI|}{|I \cup II \cup III \cup IV \cup V \cup VI|} \quad (8)$$

6.3 Similarity Functions

To make the clustering solution in Sec. 5 compatible with approximate preference relations, we extend the similarity measures, using ideas inspired by the Jaccard similarity for non-negative multidimensional real vectors [4].

1) Jaccard Similarity Consider an attribute d with $|dom(d)| = m$. For each cluster U , construct a vector $\mathbf{U} = (\mathbf{U}(1), \mathbf{U}(2), \dots, \mathbf{U}(P_2^m))$. For $i \in [1, P_2^m]$, $\mathbf{U}(i)$ represents the frequency of A_i (Definition 6.1) in U . Given two clusters U and V , their *Jaccard similarity* on attribute d is

$$sim_{ij}^d(U, V) = \frac{\sum_i \min(\mathbf{U}(i), \mathbf{V}(i))}{\sum_i \max(\mathbf{U}(i), \mathbf{V}(i))} \quad (9)$$

Example 6.8. Consider U_1 and U_3 in Table 3. Suppose $A(i)$ for $i \in [1, P_2^m]$ are ((Apple, Lenovo), (Apple, Samsung), (Apple, Toshiba), (Lenovo, Apple), (Lenovo, Samsung), (Lenovo, Toshiba), (Toshiba, Apple), (Toshiba, Lenovo), (Toshiba, Samsung), (Samsung, Apple), (Samsung, Lenovo), (Samsung, Toshiba)). The two vectors are $\mathbf{U}_1 = (2/2, 2/2, 0/2, 0/2, 2/2, 0/2, 0/2, 1/2, 2/2, 0/2, 0/2, 0/2)$ and $\mathbf{U}_3 = (0/2, 2/2, 1/2, 2/2, 2/2, 2/2, 0/2, 0/2, 1/2, 0/2, 0/2, 0/2)$. For instance, \mathbf{U}_1 has $1/2$ on the 8^{th} -dimension since only one of the two users' preference relations contains (Toshiba, Lenovo). Hence, $sim_{ij}^{\text{brand}}(U_1, U_3) = 0.36$. △

2) Weighted Jaccard Similarity This measure, denoted as sim_{wj}^d , extends the namesake measure in Sec. 5 with the idea above. Its definition is the same as Eq. 9 except that a value $\mathbf{U}(i)$ in a vector represents the frequency of A_i in U that takes into consideration the weights explained in Sec. 5. Consider A_i as the preference tuple $(A_i(x), A_i(y))$. This similarity measure is defined as follows.

$$sim_{wj}^d(U, V) = \sum_i \left(\min \left(\frac{1}{|U|} \times \sum_{c \in U} \frac{1}{\min D(s, A_i(x))+1}, \frac{1}{|V|} \times \sum_{c \in V} \frac{1}{\min D(s, A_i(x))+1} \right) \right. \\ \left. / \sum_i \left(\max \left(\frac{1}{|U|} \times \sum_{c \in U} \frac{1}{\min D(s, A_i(x))+1}, \frac{1}{|V|} \times \sum_{c \in V} \frac{1}{\min D(s, A_i(x))+1} \right) \right) \right) \quad (10)$$

Example 6.9. In Table 3, in the partial order depicting $\succ_{c_6}^{\text{brand}}$, the distance to *Apple* from the maximal value *Lenovo* is 1, i.e., the weight of *Apple* is $1/2$. Since only one of the two users in U_3 has (Apple, Toshiba) in their preference relation, \mathbf{U}_3 has $\frac{1}{2} + 0 = \frac{1}{4}$ on the 3^{rd} -dimension. In this way, we get $\mathbf{U}_1 = (2/2, 2/2, 0/2, 0/2, 1/2, 0/2, 0/2, 1/2, 2/2, 0/2, 0/2, 0/2)$ and $\mathbf{U}_3 = (0/2, 1/4, 2/2, 2/2, 2/2, 0/2, 0/2, 1/4, 0/2, 0/2, 0/2)$. Therefore, $sim_{wj}^{\text{brand}}(U_1, U_3) = 0.19$. △

7 ALIVE OBJECT DISSEMINATION

In Sec. 1, we discussed motivating applications such as social network content dissemination, news delivery and product recommendation. The significance of a particular social network content (e.g. a post in Facebook) or a piece of news diminishes eventually. Similarly, in any inventory, products are consumed and perishable products expire over time. In other words, objects can have limited lifetime. Thus, upon the arrival of a new object, it needs to compete only with the alive objects. To meet this requirement, we extend our problem as continuous monitoring of Pareto frontiers over *alive objects* for many users and formalize it as finding Pareto frontiers over *sliding window*.

Suppose $\mathcal{O} = \{o_1, o_2, \dots, o_N\}$ is a stream of objects, in which the subscript of each object is its timestamp. We consider a sliding window as a sequence of W recent objects. Upon the arrival of an incoming object o_{in} , an object o_{out} expires if $in - out = W$. Specifically, the sliding window contains objects whose timestamps are in $(out, in]$, i.e., an object $o_i \in \mathcal{O}$ is alive during $(out, in]$ if $i \in (out, in]$. Given the concept of sliding window, we extend the definition of Pareto frontier in Def. 3.3 and the problem statement in Sec. 3.

Definition 7.1 (Pareto Frontier). An alive object o is Pareto-optimal with respect to c , if no other alive object dominates it. $\mathcal{P}_c = \{o_i \in \mathcal{O} \mid \nexists o_j \in \mathcal{O} \text{ s.t. } o_j \succ_c o_i \wedge i, j \in (out, in]\}$. The target users of o_{in} is $C_{o_{in}} = \{c \in C \mid o_{in} \in \mathcal{P}_c\}$ (Def. 3.4). △

Problem Statement The problem of continuous monitoring of Pareto frontiers over sliding window is, given a set of users C , their preference relations on attributes \mathcal{D} , and a stream of objects \mathcal{O} with the incoming object o_{in} as well as the outgoing object o_{out} , find $C_{o_{in}}$ —the target users of o_{in} .

Algorithms BaselineSW and FilterThenVerifySW We extend Baseline and FilterThenVerify to BaselineSW and FilterThenVerifySW, respectively, to accommodate sliding window. We note that no prior work studied Pareto frontier maintenance with regard to strict partial orders over sliding window. [15, 16, 21] studied skyline maintenance over sliding window, assuming numeric attributes. [18] considered total orders (with ties) on categorical

attributes instead of general partial orders. There is no clear way to extend these works for partially ordered attributes.

Due to space limitations, we leave the detailed pseudo codes and descriptions of BaselineSW and FilterThenVerifySW to the extended version of this paper [20]. Below we highlight the key concepts that dictate the design of these algorithms.

Under the constraint of having a sliding window, an object can be excluded from Pareto frontier forever if it is dominated by any succeeding object. This observation is formalized as Theorem 7.2.

THEOREM 7.2. *Consider a user $c \in C$ and two objects $o_i, o_j \in O$ such that $o_i \prec_c o_j$ and $i < j$. After the arrival of o_j , o_i can never be part of \mathcal{P}_c in its remaining lifetime.* \triangle

Proof: Since $i < j$, o_i expires before o_j and the sliding window always includes o_j if it includes o . Since o_j dominates o_i , o_i will never get into \mathcal{P}_c after the arrival of o_j . \blacksquare

By Theorem 7.2, we extend our algorithms to maintain a *Pareto frontier buffer* which stores at most W recent objects that are not dominated by any succeeding object. Clearly, o_{in} is part of the Pareto frontier buffer.

Definition 7.3 (Pareto Frontier Buffer). With regard to user c and the sliding window $(out, in]$, an alive object o belongs to the Pareto frontier buffer if it is not dominated by any succeeding object. The Pareto frontier buffer is $\mathcal{P}\mathcal{B}_c = \{o_i \in O \mid \nexists o_j \in O \text{ s.t. } o_j \succ_c o_i \wedge i, j \in (out, in] \wedge i < j\}$. By definition, $\mathcal{P}\mathcal{B}_c \supseteq \mathcal{P}_c$ (Def. 7.1). \triangle

THEOREM 7.4. *Given a set of users U , for all $c \in U$, i) $\mathcal{P}\mathcal{B}_U \supseteq \mathcal{P}_U$ and ii) $\mathcal{P}\mathcal{B}_U \supseteq \mathcal{P}\mathcal{B}_c$.* \triangle

Proof: i) Together Def. 7.1 and 7.3 imply that $\mathcal{P}\mathcal{B}_U \supseteq \mathcal{P}_U$.

ii) We prove by contradiction. Suppose that there exists $c \in U$ such that $\mathcal{P}\mathcal{B}_U \not\supseteq \mathcal{P}\mathcal{B}_c$, which would mean there exists $o \in O$ such that $o \in \mathcal{P}\mathcal{B}_c$ and $o \notin \mathcal{P}\mathcal{B}_U$. That implies the existence of an $o' \in O$ such that $o' \succ_U o$ and $o' \not\succeq_c o$. However, by Def. 4.1, $o' \succ_U o$ implies $o' \succ_c o$. Therefore, the existence of o' is impossible. In conclusion, $\mathcal{P}\mathcal{B}_U \supseteq \mathcal{P}\mathcal{B}_c$. \blacksquare

Note that, BaselineSW needs to maintain an exclusive Pareto frontier buffer for each user ($\mathcal{P}\mathcal{B}_c$) while a Pareto frontier buffer per cluster ($\mathcal{P}\mathcal{B}_U$) is sufficient for FilterThenVerifySW.

8 EXPERIMENTS

8.1 Experiment Setup

The algorithms were implemented in Java. The maximal heap size of Java Virtual Machine (JVM) was set to 16 GB. The experiments were conducted on a computer with 2.0 GHz Quad Core 2 Duo Xeon CPU running Ubuntu 8.10.

Datasets Currently there exists no publicly available dataset that captures real users' preferences in partial orders. We thus simulated such partial orders using two real datasets of users' preferences.

Movie Dataset We joined the Netflix dataset (netflixprize.com) with data from IMDB (imdb.com). The Netflix dataset contains the ratings (ranging from 0 to 5) given by users to movies. From IMDB we fetched the movies' attribute values, including actors, directors, genres, and writers. In this way, we found the attributes of 12,749 Netflix movies. The goal is to, for each particular movie, identify users who may like it according to their preferences on those attributes. The mapping from our problem formulation to this dataset is the following: (i) O is the set of 12,749 movies. (ii) C is the set of users. It includes the 1,000 most active users based on how many movies they have rated. (iii) $\mathcal{D} = \{\text{actor,}$

director, genre, writer}. (iv) Given the lack of user preference data, for each attribute, the partial order corresponding to a user's preferences is simulated as follows. For two attribute values, the user's preference is based on the *average rating* and the *count* of movies satisfying these attribute values. More specifically, consider a user c who has rated m movies featuring actor a . Suppose the ratings of these movies are r_1, r_2, \dots, r_m . Given c and a , the average rating is $R_a = \frac{\sum_i r_i}{m}$ and the count is $M_a = m$. Consider another actor b . If $(R_a > R_b \wedge M_a \geq M_b) \vee (R_a \geq R_b \wedge M_a > M_b)$, then $(a, b) \in \succ_c^{\text{actor}}$. Intuitively, if user c watches more movies featuring a than b and gives them higher ratings, our simulation assumes the user prefers a to b .

Publication Dataset We collected from the ACM Digital Library (dl.acm.org) 17,598 publications and their attributes, including affiliations, authors, conferences and topic keywords. The users are the authors themselves. The goal is to notify them about newly published articles. The recommendations are based on the users' preference relations on the attributes. The mapping from our problem formulation to this dataset is the following: (i) O is the set of papers. (ii) C is the set of authors. It includes the 1,000 most prolific authors based on how many publications they have, similar to the 1,000 most active users in the movie dataset. (iii) $\mathcal{D} = \{\text{affiliation, author, conference, keyword}\}$. The domain of attribute author is the same 1,000 authors in C . (iv) Given a user, the partial order on each attribute is simulated based on their preferences on the attribute values. The preference between two values on affiliation (and similarly author) is based on the *number of collaborations* between the user and the affiliation/author and the *number of citations*. For conference and keyword, the preference between two values is based on *number of publications* and *number of citations*. More specifically, consider a user c and an affiliation (or similarly another author) a . Suppose c has p_a collaborations with a and has cited articles from a q_a times. If $(p_a > p_b \wedge q_a \geq q_b) \vee (p_a \geq p_b \wedge q_a > q_b)$, then $(a, b) \in \succ_c^{\text{affiliation}}$ (or $(a, b) \in \succ_c^{\text{author}}$). With regard to a conference (keyword) x , suppose c has r_x publications associated with x and has cited publications associated with x s_x times. If $(r_x > r_y \wedge s_x \geq s_y) \vee (r_x \geq r_y \wedge s_x > s_y)$, then $(x, y) \in \succ_c^{\text{conference}}$ (or $(x, y) \in \succ_c^{\text{keyword}}$).

8.2 Baseline, FilterThenVerify, and FilterThenVerifyApprox

We conducted experiments to compare the performance of Baseline, FilterThenVerify and FilterThenVerifyApprox. For FilterThenVerify (resp. FilterThenVerifyApprox), users are clustered by the conventional hierarchical agglomerative clustering algorithm [9] using the similarity functions in Sec. 5 (resp. Sec. 6.3) and, for each cluster, it extracts the common preference relation (resp. approximate common preference relation). The experiments use three parameters which are number of objects ($|O|$), number of attributes (d), and branch cut (h). In hierarchical clustering, the *branch cut* h is a threshold that controls the number of clusters by governing the minimum pairwise similarity that two clusters must satisfy in order to be merged into one cluster. The sequential order of merging clusters is depicted as a tree called *dendrogram*. The branch cut thus controls where to cut the dendrogram. In Example 5.5, the set of clusters are $\{\{c_1, c_2, c_5, c_6\}, \{c_3, c_4\}\}$ for $h \in (0, \frac{3}{11}]$. This is because $\text{sim}(U_4, U_2) = 0$ where $U_2 = \{c_3, c_4\}$ and U_4 is the cluster composed of c_1, c_2, c_5 , and c_6 .

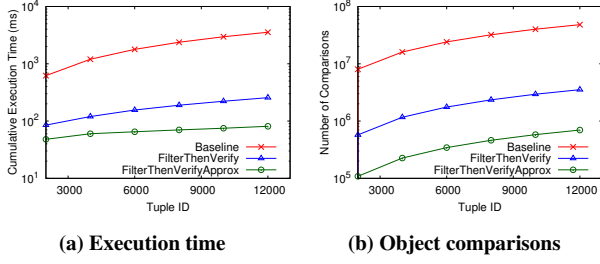


Figure 3: Comparison of Baseline, FilterThenVerify and FilterThenVerifyApprox on the movie dataset. Varying $|O|$, $h = 0.55$, $d = 4$.

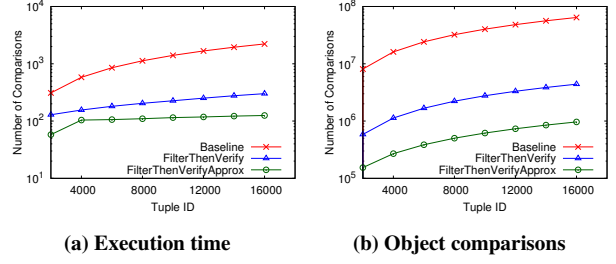


Figure 4: Comparison of Baseline, FilterThenVerify and FilterThenVerifyApprox on the publication dataset. Varying $|O|$, $h = 0.55$, $d = 4$.

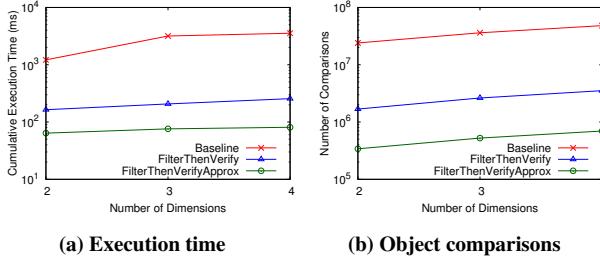


Figure 5: Comparison of Baseline, FilterThenVerify and FilterThenVerifyApprox on the movie dataset. Varying d , $|O| = 12, 749$, $h = 0.55$.

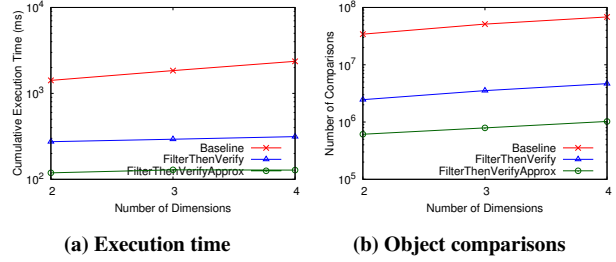


Figure 6: Comparison of Baseline, FilterThenVerify and FilterThenVerifyApprox on the publication dataset. Varying d , $|O| = 17, 598$, $h = 0.55$.

Dataset	$ O $	$h = 0.70$			$h = 0.65$			$h = 0.60$			$h = 0.55$		
		Precision	Recall	F-measure	Precision	Recall	F-measure	Precision	Recall	F-measure	Precision	Recall	F-measure
Movie	12, 749	100	95.43	97.67	100	93.93	96.87	99.99	93.28	96.52	99.99	90.46	94.99
Publication	17, 598	100	96.59	98.27	100	95.85	97.88	100	95.54	97.72	100	95.13	97.51

Table 7: The precision, recall and F-measure (in percentage) of FilterThenVerifyApprox. Varying h , $d=4$.

Fig.3a shows, for each of the three methods on the movie dataset, how its cumulative execution time (by milliseconds, in logarithmic scale) increases while the objects (i.e., movies) are sequentially processed. Fig.4a depicts similar behaviours of these methods on the publication dataset. Fig.3b and Fig.4b, for the two datasets separately, further present the amount of work done by these methods, in terms of number of pairwise object comparisons (in logarithmic scale) for maintaining Pareto frontiers. The figures show that FilterThenVerify and FilterThenVerifyApprox beat Baseline by 1 to 2 orders of magnitude. The reason is as follows. With regard to a user c , Baseline considers all objects as candidate Pareto-optimal objects and compares all pairs. On the contrary, FilterThenVerify eliminates an object o if the corresponding common preference tuples disqualify o . FilterThenVerifyApprox incurs even less comparisons by benefiting from shared computations for clusters of users.

Fig.5a (Fig.6a) shows that the execution time of all these methods increased super-linearly by number of attributes (d). Fig.5b (Fig.6b) further reveals that the number of object comparisons also increases similarly. This is not surprising because more attributes result in larger Pareto frontiers, which makes it necessary for objects to be compared with more existing Pareto-optimal objects.

Table 7 reports the precision, recall and F-measure of FilterThenVerifyApprox on varying h . We can observe that, when h got smaller, the recall slowly decreased. This is expected because smaller h results in larger clusters and potentially more approximate common preference tuples for each cluster. Those approximate common preference tuples cause false negatives—the domination and elimination of objects that are instead in the Pareto frontier under the true common preference tuples, which

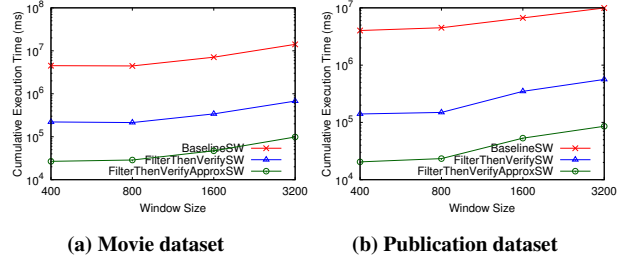


Figure 7: Effect of window size. Varying W , $|O| = 1M$, $h = 0.55$, $d = 4$.

are a subset of the approximate common preference tuples. What can be more surprising is the almost perfect precision under the various h values in Table 7, i.e., almost no false positives were introduced into the results. For a user c , an object o becomes a false positive if every single Pareto optimal object that dominates o becomes a false negative. As long as one of its dominating objects is not mistakenly filtered out, o will not be mistakenly introduced into the Pareto frontier. Therefore, an object is much less likely to become a false positive than a false negative. Overall, under the h values in Table 7, both precision and recall remain high. This may suggest that the thresholds θ_1 and θ_2 (Sec. 6.1) effectively ensure that the approximate common preference relation only includes frequent preference tuples and does not overgrow in size.

8.3 BaselineSW, FilterThenVerifySW, and FilterThenVerifyApproxSW

We further compare the performance of FilterThenVerifySW and FilterThenVerifyApproxSW with BaselineSW. In this regard, we simulated two data streams—movie and publication where O is

Data stream	W	$h = 0.70$			$h = 0.65$			$h = 0.60$			$h = 0.55$		
		Precision	Recall	F-measure	Precision	Recall	F-measure	Precision	Recall	F-measure	Precision	Recall	F-measure
Movie	400	100	89.36	94.38	100	87.33	93.24	100	85.94	92.44	100	81.95	90.08
	800	100	87.87	93.54	100	85.78	92.34	100	84.04	91.33	100	80.10	88.95
	1600	100	88.65	93.98	100	86.58	92.81	100	85.01	91.90	100	81.10	89.56
	3200	99.99	94.80	97.33	100	93.08	96.41	100	92.29	95.99	100	88.99	94.17
Publication	400	100	94.58	97.21	100	93.57	96.68	100	92.98	96.36	100	92.06	95.87
	800	100	94.79	97.32	100	93.60	96.70	100	93.01	96.38	100	91.98	95.82
	1600	100	94.62	97.24	100	93.44	96.61	100	92.85	96.29	100	91.81	95.73
	3200	100	96.71	98.33	100	95.98	97.95	100	95.67	97.79	100	95.27	97.58

Table 8: The precision, recall and F-measure (in percentage) of FilterThenVerifyApproxSW. Varying W and h , $|O|=1M$, $d=4$.

composed of duplicated sequence of the corresponding dataset such that $|O|=1$ million. Following [21], we experimented with windows of size 400, 800, 1,600, and 3,200. Fig.7a shows the cumulative execution times (by milliseconds, in logarithmic scale) of the aforementioned methods on the movie stream. Fig.7a reveals that the cumulative execution times increase super-linearly by W as wider window broadens the size of Pareto frontiers. These figures illustrate that both FilterThenVerifySW and FilterThenVerifyApproxSW outperformed BaselineSW by 1 to 2 orders of magnitude, which concurs with the comparative behaviours of FilterThenVerify, FilterThenVerifyApprox and Baseline. This concurrence is also applicable for the publication stream (Fig.7b). The reason behind the comparative behaviour of Baseline, FilterThenVerify and FilterThenVerifyApprox is also applicable in this case. Moreover, BaselineSW maintains exclusive Pareto buffer for each user (\mathcal{PB}_C) while FilterThenVerifySW shares a Pareto buffer across users in a cluster (\mathcal{PB}_U). Therefore, in sliding window protocol, the filter-then-verify approach attains the benefit of clustering in a greater extent.

Table 8 demonstrates the precision, recall and F-measure of FilterThenVerifyApproxSW on varying W and h . We can observe that the recall declines slowly by h . Yet h does not have significant impact on the efficacy of FilterThenVerifyApproxSW. Besides, the loss of accuracy is due to false negatives rather than false positives. These behaviors concur with FilterThenVerifyApprox and the reasons behind are same as before. In addition, Table 8 reveals that W does not have noticeable impact on efficacy and FilterThenVerifyApprox remains effective on varying W .

9 CONCLUSION

We studied the problem of continuous object dissemination, which is formalized as finding the users who approve a new object in Pareto-optimality. We designed algorithm for efficient finding of target users based on sharing computation across similar preferences. To recognize users of similar preferences, we studied the novel problem of clustering users where each user's preferences are described as strict partial orders. We also presented an approximate solution of the problem of finding target users, further improving efficiency with tolerable loss of accuracy. Experimental evaluation validated the efficiency and effectiveness of our proposed solutions.

Acknowledgements: The work is partially supported by NSF grant IIS-1719054. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the funding agencies. Furthermore, we thank Fatma Arslan for her contribution in data collection.

REFERENCES

[1] O. Barndorff-Nielsen and M. Sobel. On the distribution of the number of admissible points in a vector random sample. *Theory of Probability & Its Applications*, 11(2), 1966.

[2] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, 2001.

[3] C.-Y. Chan, P.-K. Eng, and K.-L. Tan. Stratified computation of skylines with partially-ordered domains. In *SIGMOD*, 2005.

[4] F. Chierichetti, R. Kumar, S. Pandey, and S. Vassilvitskii. Finding the jaccard median. In *SODA*, 2010.

[5] J. Chomicki. Preference formulas in relational queries. *TODS*, 28(4), 2003.

[6] E. Dellis and B. Seeger. Efficient computation of reverse skyline queries. In *VLDB*, 2007.

[7] R. Fagin. Combining fuzzy information from multiple systems. In *PODS*, 1996.

[8] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.

[9] J. Han, M. Kamber, and J. Pei. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 3rd edition, 2011.

[10] T. Kamishima and S. Akaho. Clustering orders. In *Discovery Science*, 2003.

[11] T. Kamishima and S. Akaho. Efficient clustering for orders. In *Mining Complex Data*. 2009.

[12] W. Kießling. Foundations of preferences in database systems. In *VLDB*, 2002.

[13] H. T. Kung, F. Luccio, and F. P. Preparata. On finding the maxima of a set of vectors. *Journal of ACM*, 22(4), Oct. 1975.

[14] K. C. Lee, B. Zheng, H. Li, and W.-C. Lee. Approaching the skyline in z order. In *VLDB*, 2007.

[15] X. Lin, Y. Yuan, W. Wang, and H. Lu. Stabbing the sky: Efficient skyline computation over sliding windows. In *ICDE*, 2005.

[16] M. Morse, J. M. Patel, and W. I. Grosky. Efficient continuous skyline computation. *Information Sciences*, 177(17), 2007.

[17] D. Sacharidis, S. Papadopoulos, and D. Papadias. Topologically sorted skylines for partially ordered domains. In *ICDE*, 2009.

[18] N. Sarkas, G. Das, N. Koudas, and A. K. Tung. Categorical skylines for streaming data. In *SIGMOD*, 2008.

[19] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl. Item-based collaborative filtering recommendation algorithms. In *WWW*, 2001.

[20] A. Sultana and C. Li. Continuous monitoring of pareto frontiers on partially ordered attributes for many users. *CoRR*, abs/1709.08312, 2017.

[21] Y. Tao and D. Papadias. Maintaining sliding window skylines on data streams. *TKDE*, 18(3), 2006.

[22] A. Ukkonen. Clustering algorithms for chains. *The Journal of Machine Learning Research*, 12, 2011.

[23] A. Vlachou, C. Doukeridis, Y. Kotidis, and K. Norvag. Reverse top-k queries. In *ICDE*, 2010.

[24] A. Vlachou, C. Doukeridis, K. Nøravåg, and Y. Kotidis. Branch-and-bound algorithm for reverse top-k queries. In *SIGMOD*, 2013.

[25] R. C.-W. Wong, A. W.-C. Fu, J. Pei, Y. S. Ho, T. Wong, and Y. Liu. Efficient skyline querying with variable user preferences on nominal attributes. *VLDB*, 1(1), 2008.

[26] R. C.-W. Wong, J. Pei, A. W.-C. Fu, and K. Wang. Mining favorable facets. In *SIGKDD*, 2007.

[27] R.-W. Wong, J. Pei, A.-C. Fu, and K. Wang. Online skyline analysis with dynamic preferences on nominal attributes. *TKDE*, 21(1), 2009.

[28] P. Wu, D. Agrawal, O. Egecioglu, and A. El Abbadi. Deltasky: Optimal maintenance of skyline deletions without exclusive dominance region generation. In *ICDE*, 2007.

[29] A. Yu, P. K. Agarwal, and J. Yang. Processing a large number of continuous preference top-k queries. In *SIGMOD*, 2012.

[30] S. Zhang, N. Mamoulis, D. W. Cheung, and B. Kao. Efficient skyline evaluation over partially ordered domains. *VLDB*, 3(1-2), 2010.

Optimizing Selection Processing for Encrypted Database using Past Result Knowledge Base

Wai Kit Wong
Hang Seng Management College
wongwk@hsmc.edu.hk

Kwok Wai Wong
Hang Seng Management College
mkwai2016@gmail.com

Ho-Yin Yue
Hang Seng Management College
willyyue@hsmc.edu.hk

ABSTRACT

Data confidentiality is concerned in database-as-a-service (DBaaS) model. The cloud server should not have access to user's plain data. Data is encrypted before they are stored in cloud database. Query computation over encrypted data by the server is not straight-forward. Many research works have been done on this problem. A common goal is to let the server obtain the selection result without leaking information about plain data. In existing solutions, the selection result is simply dumped by the server after the query answer is returned. Our idea is to make use of such *past results* of selections to improve processing speed for new queries. We developed an indexing mechanism called *past result knowledge base* (PRKB) to improve processing speed of selection with comparison predicate(s) in EDBMS. All operations related to PRKB are done by the server only. In our empirical studies, PRKB can reduce processing cost by orders of magnitudes compared to the case PRKB is not used.

1 INTRODUCTION

In database-as-a-service (DBaaS) model, a data owner (DO) uploads its data to a database managed by a third party service provider (SP) who is responsible to answer DO's queries and provides administrative services, e.g., backup recovery and access control. Data confidentiality is concerned when SP is compromised, e.g., by a malicious DBA administrating the database server. For instance, a rogue DBA has stolen 2.3 millions customer records of a Fortune 500 company¹, including bank account and credit card information. Encrypted database management systems (EDBMSs), such as Cipherbase [2–4] and SDB [19, 35], are recently developed to address data confidentiality concerns in case SP is compromised. The idea in EDBMS is to use *application level encryption* where data is encrypted and decrypted by DO and the private keys are only known to DO. Even if an attacker somehow gets access to the database at SP, the attacker can only obtain encrypted information without the keys to decrypt the data.

A challenge in EDBMS is to allow SP to compute selection over encrypted data, without knowing any other information about plain data (so that minimal number of encrypted tuples are processed in the next operations, e.g., join and/or aggregation). Many solutions were proposed, e.g., [12, 25] for range query and [13] for keyword search. In this paper, we address optimization of computing selection with comparison predicate(s) in EDBMS.

Fig. 1 shows an overview of our method, *past result knowledge base* (PKRB). The results of past selections are consolidated and stored in PRKB at SP. SP can use PRKB to reduce processing cost

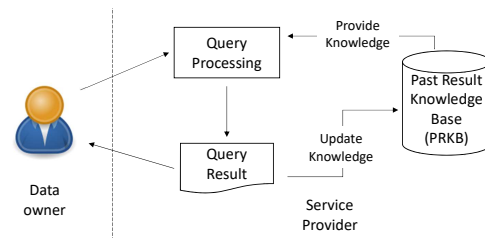


Figure 1: Overview of PRKB for query processing

Encrypted tuples	Selection		
	σ_1	σ_2	σ_3
\bar{t}_1	✓	X	✓
\bar{t}_2	X	✓	✓
\bar{t}_3	✓	✓	?
\bar{t}_4	✓	✓	?

Table 1: Example scenario. If an encrypted tuple satisfies (or does not satisfies resp.) a selection, a '✓' (or 'X' resp.) is shown. '?' denotes that the result for the encrypted tuple is not known yet.

of a new selection. We use the following example to illustrate the cost reduction idea.

Suppose there are 4 encrypted tuples \bar{t}_i for $i = 1$ to 4. Each tuple has one attribute X only. There are 3 selections σ_1 , σ_2 , and σ_3 , each with a simple comparison predicate in the form of ' $X < c$ ' or ' $X > c$ ' where c is a user-defined parameter unknown to SP. σ_1 and σ_2 are computed already. σ_3 is partially computed. The scenario is shown in Table 1.

SP can reason as following to determine the values of '?' in Table 1. The result of σ_1 partitions the encrypted tuples into two groups, $P_1 = \{\bar{t}_1, \bar{t}_3, \bar{t}_4\}$ and $P_2 = \{\bar{t}_2\}$. We use $P_i > P_j$ to denote that all tuples in P_i have a larger plain value than all tuples in P_j . Since σ_1 is a simple comparison predicate, there are only two possible scenarios: either (i) $P_1 > P_2$ or (ii) $P_2 > P_1$. We assume it is the case of scenario (i), i.e., \bar{t}_2 has the smallest plain value. Similarly, from the result of σ_2 , SP obtains two partitions $P_3 = \{\bar{t}_1\}$ and $P_4 = \{\bar{t}_2, \bar{t}_3, \bar{t}_4\}$. It must be either $P_3 > P_4$ or $P_4 > P_3$. Since $\bar{t}_2 \in P_4$ and \bar{t}_2 has the the smallest value, it must be the case $P_3 > P_4$. As a result, SP obtains the order of (plain values of) all encrypted tuples in this scenario as ' $\bar{t}_2 < \bar{t}_3, \bar{t}_4 < \bar{t}_1$ '. This order information is *partial* only as SP cannot determine which encrypted tuple is larger for \bar{t}_3 and \bar{t}_4 . For σ_3 , \bar{t}_1 and \bar{t}_2 are found to satisfy the selection condition. \bar{t}_3 and \bar{t}_4 are ordered between \bar{t}_1 and \bar{t}_2 , so \bar{t}_3 and \bar{t}_4 must also satisfy the selection condition, i.e., both '?' must be '✓' in Table 1. SP can perform the same analysis for scenario (ii) and obtains the same conclusion.

In the above example, SP can determine whether \bar{t}_3 and \bar{t}_4 satisfy a new selection σ_3 without accessing them and thus saves the processing cost on these two encrypted tuples. Such saving is significant because the process to check whether an encrypted

¹<http://www.computerworld.com/article/2542360/security0/database-admin-steals-2-3m-consumer-records-at-fidelity-national-subsiary.html>

tuple satisfies a comparison predicate is usually expensive. For example, in Cipherbase [2], the encrypted tuple is decrypted (within a trusted hardware) before the predicate is tested. The additional decryption cost is significant compared to the cost of a simple comparison. As demonstrated in our empirical evaluation, only a small portion of encrypted tuples cannot be determined using PRKB and only this group of tuples require to be processed by SP using the usual cryptographic way. The overall processing cost is greatly reduced.

We highlight the distinguished features of PRKB as follows:

- *DO is not involved* in any part (e.g., building or using) of PRKB. No information is required to be sent from DO to SP for PRKB. All information in PRKB is solely based on what SP has observed in past query computation. It can be realized easily that PRKB does not leak more information than the underlying EDBMS.
- *PRKB is compatible to any encryption scheme* that tells SP which encrypted tuples satisfy the selection. As long as the encryption scheme provides such trapdoor, PRKB can be used on top of it. This allows PRKB to be deployed on top of many existing systems, e.g., Cipherbase and SDB.
- *Information in PRKB is in plain*. Unlike encrypted index, PRKB consists of information about past selection results. All operations related to PRKB are efficient and the size of PRKB is compact.

The rest of this paper is organized as follows. We discuss related work in Sec. 2. In Sec. 3, we define the models used in our problem. We describe what PRKB is and how SP builds PRKB in Sec. 4. In Sec. 5 and Sec. 6, we describe our algorithms for processing a single comparison predicate and multi-dimensional range query respectively. We describe how to handle database update in Sec. 7. We empirically evaluate PRKB and related algorithms in Sec. 8. We conclude this paper in Sec. 9.

2 RELATED WORK

Different solutions were developed for computing individual database operations over encrypted data, e.g., range query [6, 12, 25, 28], keyword search [7, 13] and join [26]. A potential problem of these solutions is that integration of the above solutions is not trivial. Encrypted database management system (EDBMS) [2, 4, 5, 19, 29, 33, 35] offers an integrated solution that supports a wide range of SQL operations. We aim to deploy our optimization technique for selection with comparison predicate(s) in EDBMS. We first review existing EDBMSs in Sec. 2.1. Then, we review indexing options for EDBMS in Sec. 2.2. Lastly, we review techniques that hide selection results from SP in Sec. 2.3.

2.1 EDBMS

There are several approaches to implementing EDBMS.

The first approach, e.g., TrustedDB [5] and Cipherbase [2, 4], makes use of a trusted hardware. There is a trusted machine (TM) at SP. For instance, TrustedDB uses Cryptographic Coprocessor and Cipherbase uses FPGA. Such hardware devices are (physically) tamper-resistant. An attacker is assumed not to be able to see the data or process inside TM. TM is given the decryption key of DO. Any computation related to encrypted data can be handled by TM. For instance, to process a comparison predicate, the encrypted value of each tuple and the instructions (with encrypted user parameters) are passed to TM. TM decrypts the data, makes the comparison, and returns the comparison result to SP.

The second approach uses secret sharing methods, e.g., SDB [19, 35]. Secret sharing splits each data item into shares. Some shares are stored at DO² while some are stored at SP. Without collecting all the shares of a data item, SP cannot recover its plain value. Multi-party computation (MPC) operators are developed to execute database operations by DO and SP communicating with each other in multiple rounds. An advantage of MPC approach is that any computation can be computed [15]. However, it incurs a high communication cost in query processing.

Another approach is to use multiple encryption schemes, each to support a different set of operations. Example system is CryptDB [29] / MONOMI [33]. (MONOMI is an extension of CryptDB to further support aggregation.) Specifically, CryptDB uses order preserving encryption (OPE), e.g., [1, 28], to process comparison predicates. OPE preserves the numerical order of plain data under encryption, i.e., if $x > y$, $E(x) > E(y)$ for any x, y where E denotes the encryption function. Comparison predicates can be computed efficiently and indexing over OPE-encrypted data is just like indexing over plain data. A downside of this approach is that it leaks the total order of plain data to SP. Recent studies show that inference attack [22, 27] can recover accurately the plain data of OPE-encrypted data using the total order information.

2.2 Indexing on encrypted data

Our method, PRKB, is similar to indexing, because SP uses additional space to remember past results in order to boost the performance in query computation. PRKB has a significant difference to mainstream indexing methods for encrypted data: *PRKB is solely done by SP*, while existing indexing mechanisms for encrypted data require DO's involvement, e.g., to build the encrypted index, or requires DO to encrypt data using specific algorithms. In the following, we briefly discuss existing indexing methods for encrypted data.

In [11, 14, 31], an encrypted index tree is built by DO and stored at SP. SP simply serves as a storage without processing capabilities. DO retrieves parts of the index from SP iteratively to traverse the index. Data confidentially can be proven as SP never see any data in plain. Access pattern of the index can also be protected from SP, e.g., in [14], at the cost of increased processing and communication cost. DO has a significant amount of workload. We do not prefer this approach in DBaaS where DO may not be as powerful as SP.

[12, 23, 25] developed new encryption methods for computing comparison predicate or range query with indexing support. In [23], data is encrypted using a special vector encryption method [34]. An index can be built over these encrypted values. A problem is there is information leaked in the encryption scheme, as shown in [17], and plain data can be recovered in some scenarios. In [25], a security notion of index indistinguishable is introduced and an index is developed that achieves the proposed security notion. However, such security notion is proven to be weak [10]. In [12], it transforms the problem of range query into keyword search and uses existing searchable symmetric encryption (SSE) [7, 8] to compute the result. A series of schemes with different indexing options, each offers different security strength, is developed. In all methods above, SP see the selection results as this is the objective of the problem. Our method PRKB can also be implemented on top of these encryption methods. In our empirical studies, we will compare our method to [12].

²In SDB, the shares at DO can be generated using an RSA-like share-generating function. This reduces storage cost at DO.

Some EDBMSs we discussed in Sec. 2.1 also use indexing. Cipherbase uses an encrypted B+-tree. The index reveals the total order of plain data to SP and is thus also vulnerable to inference attack. SDB uses domain-partitioning index [18, 21]. The data domain is divided into partitions by DO. SP is informed by DO the partition each data item falls into. Due to additional information leak, we do not consider these methods suitable for our problem.

2.3 Hiding selection result from SP

Our problem assumes the selection result is known by SP. The selection result can potentially be hidden by access pattern hiding technique, e.g., oblivious RAM (ORAM) [16, 32] and/or private information retrieval [9]. For instance, [4] discussed an option to integrate ORAM in Cipherbase. The trade-off is that each data access has a polylog cost. Due to high overhead, ORAM was not implemented in Cipherbase. Similar to Cipherbase, other EDBMSs we reviewed in Sec. 2.1 do not use any access pattern hiding technique. Our method can be deployed on existing EDBMSs, including TrustedDB, CipherBase and SDB.

3 MODELS

3.1 Preliminary: EDBMS

Our problem is based on an underlying encrypted database management system (EDBMS). In this section, we describe the EDBMS model, that is compatible to EDBMSs that we discussed in Sec. 2.1.

Parties. There are two parties: data owner (DO) and service provider (SP). DO has a set of relational tables in the database. Each table T is a set of tuples, i.e., $T = \{t_i\}$ where t_i denotes the i -th tuple. DO encrypts T to be \bar{T} such that $\bar{T} = \{\bar{t}_i \mid \bar{t}_i = E(t_i)\}$ for every $t_i \in T$ where E denotes the encryption function. (We use \bar{X} to represent the encrypted version of X in the rest of the paper.) \bar{T} is sent to SP for storage and DO does not store T . The private keys are only known by DO.

Selection processing. EDBMS allows selection to be computed over encrypted data by SP. The selection contains one or more comparison predicates, e.g., ' $X > 10$ '. SQL supports a wide range of comparison predicates, e.g., comparison operators ($>$, $<$, \geq , and \leq), and BETWEEN operator etc. In general, for existing EDBMSs that employ attributed-based encryption, SP can tell (i) which type of operator is used because the algorithms to process them are different³; and (ii) which encrypted attribute is concerned so that other encrypted values of other attributes are not accessed during selection processing. In our problem, we focus on comparison predicate in the form of ' $X \text{ op } c$ ' where op is a comparison operator. In Appendix A, we briefly discuss how BETWEEN operator can be handled. As we discussed in Sec. 2.1, there are different ways to implement EDBMS to support selection processing. An ideal method allows SP to observe the selection result without seeing any information about plain data. We use the following model (based on the model in [24]) to capture the selection processing mechanism of EDBMS.

QPF model. Let p_i, \bar{p}_i be the plain and encrypted version of a comparison predicate. There is a query processing function (QPF) Θ such that

$$\Theta(\bar{p}_i, \bar{t}_j) = \begin{cases} 1, & \text{if } t_j \text{ satisfies } p_i \\ 0, & \text{otherwise} \end{cases}$$

³Comparison operators ($>$, $<$, \geq , and \leq) are handled by the same algorithm and hence SP cannot distinguish them.

\bar{p}_i is generated by DO and acts as a trapdoor that allows SP to observe the selection result of p_i . SP cannot observe the selection result of any predicate without such trapdoor, i.e., SP's knowledge of selection results is limited by *number of comparison predicates* issued by DO.

Applications. The above QPF model is generic [24] such that the selection processing mechanisms of majority of related work can be captured. For instance, among the EDBMSs that we have studied in Sec. 2.1, TrustedDB [5], Cipherbase [2, 4], and SDB [19, 35] satisfy this model, i.e., our method can be implemented on top of these systems. CryptDB [29] and MONOMI [33] also satisfy our EDBMS model, but they also reveal the total order of plain data (in addition to selection results) to SP. With total order known, there are simpler options to optimize query processing and security strength is significantly lowered. CryptDB and MONOMI are not our target applications and our underlying EDBMS should not reveal the total order information.

Our method can also be integrated to many other standalone methods for selection processing on encrypted database, e.g., [12, 20, 25, 30]. As long as these methods reveal the selection results to SP, our method can be applied. Methods that do not reveal the selection results to SP (see Sec. 2.3) are not our target applications. In our empirical studies, we will compare our method to the indexing method in [12], the state-of-the-art method for range query processing on encrypted data.

3.2 Problem in this paper: optimizing selection with comparison predicate

Our objective is to reduce the processing cost of selection at SP. A baseline method for SP is to test all encrypted tuples using the QPF one by one. The bottleneck of performance is QPF evaluation. Note that a comparison can be done extremely fast, e.g., in one cycle. QPF evaluation is relatively more expensive in general. For example, in Cipherbase, the encrypted tuple is decrypted within a trusted machine before the comparison is done. The decryption cost is significant compared to the simple comparison. Our goal is to *reduce number of QPF uses*.

Our problem is similar to an indexing problem, which trades (SP's) storage for speed. However, unlike traditional indexing problem for encrypted data, our method (i) does not rely on specific encryption method; and (ii) does not require DO's involvement⁴ in building and using "our index".

Our indexing mechanism, namely past result knowledge base (PRKB), composes of the following 4 algorithms.

$I \leftarrow \text{initPRKB}(\bar{T})$: is run by SP to initialize the index I .

$I \leftarrow \text{updatePRKB}(I, \bar{p}_i)$: is run by SP to update the index I with an encrypted predicate \bar{p}_i .

$(\bar{T}_W, \bar{T}_{NS}) \leftarrow \text{QFilter}(\bar{T}, I, \bar{p}_i)$: is run by SP to find two exclusive subsets of \bar{T} using the index: (i) \bar{T}_W represents the 'Winner' group. All encrypted tuples in this group must satisfy the plain predicate p_i ; and (ii) \bar{T}_{NS} represents the 'Not sure' group. Encrypted tuples in this group may satisfy p_i but some may be false positives. This group of encrypted tuples requires further processing by SP to confirm the exact selection result.

⁴DO may still be involved in QPF evaluation, e.g., in SDB [19, 35], and our method does invoke QPF. We do not count this as DO's involvement for the index because such involvement exists in EDBMS without our index.

$\overline{T_{WNS}} \leftarrow \text{QScan}(\overline{T_{NS}}, \mathcal{I}, \overline{p_i})$: is run by SP to confirm the exact selection result by examining each encrypted tuple one by one. QScan is similar to a linear scan but with optimization using information from PRKB.

SP can initiate PRKB (for an attribute) by *initPRKB* to create an ‘empty’ knowledge base as there is no past result observed by SP yet. As SP receives queries from DO, SP observes new selection results and can use them to extend PRKB using *updatePRKB*.

After executing *QFilter* and *QScan*, the selection result is then $\overline{T_W} \cup \overline{T_{WNS}}$. Fig.2b shows the procedure and messages in communication of EDBMS using PRKB. In contrast, Fig.2a shows the same procedure of EDBMS without using PRKB. As we will show in the paper, *QFilter* is cheap and $\overline{T_{NS}}$ is significantly smaller than \overline{T} . Only $\overline{T_{NS}}$ is processed by QScan. The overall cost can be thus reduced.

3.3 Security discussions

In the security model of EDBMS, an attacker has compromised SP and is able to observe anything SP can see. Or in simpler interpretation, SP is the attacker. An ideal situation is that SP observes no information about plain data. Unfortunately, there is some inherit information leak that we cannot avoid. The selection results and any information derived from them can be seen by SP because it is the objective of selection processing over encrypted data. The security goal of EDBMS is then to minimize leakage of any other information about plain data.

In our problem, we use the same attack model as EDBMS that the attacker has compromised SP. The security goal is to minimize additional leakage to SP caused by our method. As we will show in the paper, *our indexing method PRKB is built and used solely by SP using existing selection results*. No (encrypted or plain) information is ever sent from DO to SP for our index. For instance, readers can compare Fig.2b and Fig.2a which show the communication between DO and SP in EDBMS with or without using PRKB. The messages in the two cases are identical. Any information that can be derived by SP from PRKB can also be obtained by SP in EDBMS without PRKB. There is no additional leakage caused by PRKB.

Another important issue is that selection results are assumed to be observed by SP. We remark that this assumption is held in many existing methods. As discussed in [24] that selection results allow SP to eventually recover a total order of encrypted data on their plain values. The total order information can then be used in inference attack [22, 27] to recover accurate plain values. Data confidentiality is completely lost. The technique in [24] requires SP to observe $O(D^4)$ queries, where D is the domain size of an attribute, so as to let SP recover the total order information. Experiments in [24] showed that the total order can be recovered in a short time for a small data domain (e.g., $D \leq 365$). On the other hand, when the domain size D is large, it becomes impractical for SP to collect $O(D^4)$ queries for the attack. Yet, SP is able to observe certain number of queries. This allows SP to observe *partial* ordering information. We thus performed empirical evaluation to see how much ordering information SP can recover when SP observes limited number of queries. The details are presented in Sec. 8.1.

4 BUILDING PAST RESULT KNOWLEDGE BASE (PRKB)

In this section, we present what information SP can observe in query processing of EDBMS and how SP can use the observed

information to build a past result knowledge base (PRKB). PRKB can then be used by SP to reduce query processing cost for new range queries.

Consider a comparison predicate p_C in the form of ‘ $C \text{ op } x$ ’ where x is a user-defined query parameter and op is one of the following: $>$, $<$, \geq , and \leq . $\overline{p_C}$ is a trapdoor generated by DO that allows SP to observe, using QPF Θ , whether an encrypted tuple satisfies the predicate without seeing the plain data and plain predicate. Note that SP does not know which comparison operator op is used in p_C . SP can divide the encrypted tuples in \overline{T} into two partitions: (i) P_T : the partition of encrypted tuples where QPF outputs 1, i.e.,

$$P_T = \{\overline{t_i} \mid \overline{t_i} \in \overline{T} \text{ and } \Theta(\overline{p_C}, \overline{t_i}) = 1\}$$

and (ii) P_F : the partition of encrypted tuples P_F where QPF outputs 0, i.e.,

$$P_F = \{\overline{t_i} \mid \overline{t_i} \in \overline{T} \text{ and } \Theta(\overline{p_C}, \overline{t_i}) = 0\}$$

We can easily prove that either all encrypted tuples in P_T have a larger plain value on C than all encrypted tuples in P_F or it is the reverse case. For example, consider a comparison predicate ‘ $C < 9$ ’. P_T contains all encrypted tuples with plain values on C less than 9. P_F contains all encrypted tuples with plain values on C greater than or equal to 9. All encrypted tuples in P_F have a larger plain value on C than any encrypted tuple in P_T . Note that if the comparison predicate is ‘ $C > 9$ ’, all encrypted tuples in P_T have a larger plain value on C . SP cannot distinguish which set of encrypted tuples, P_T or P_F , is having larger plain values than the other. To capture the above special ordering relationship between two sets of encrypted tuples, we define two symbols below.

Definition 4.1. (Relationship between partitions.) Let $P_1 = \{\overline{t_i}\}$ and $P_2 = \{\overline{t_j}\}$ be two sets of encrypted tuples, $t_i[C]$ be the plain value of t_i on attribute C . We write $P_1 \overset{C}{<} P_2$ if $\forall \overline{t_i} \in P_1, \forall \overline{t_j} \in P_2, t_i[C] < t_j[C]$. We write $P_1 \overset{C}{\mapsto} P_2 \overset{C}{\mapsto} \dots \overset{C}{\mapsto} P_n$ if either (i) $P_1 \overset{C}{<} P_2 \overset{C}{<} \dots \overset{C}{<} P_n$ or (ii) $P_1 \overset{C}{>} P_2 \overset{C}{>} \dots \overset{C}{>} P_n$.

If the comparison predicate is ‘ $C < 9$ ’, $P_T \overset{C}{<} P_F$. However, SP does not observe the plain comparison predicate. SP can only conclude that $P_T \overset{C}{\mapsto} P_F$. The ordering information SP can learn from $\overline{p_C}$ is *partial* only because (i) SP does not know the ordering relationship between individual tuples in a partition; and (ii) SP cannot conclude which partition of P_T and P_F is actually larger.

Definition 4.2. (Partial order partitions of a relational table.) Let \overline{T} be a set of encrypted tuples. We define partial order partitions of \overline{T} , denoted as POP_k^C , as a set of k partitions $P_i \subset \overline{T}$ s.t. (i) $P_i \cap P_j = \emptyset$ for $i \neq j$; (ii) $\bigcup_{i=1}^k P_i = \overline{T}$; and (iii) $P_1 \overset{C}{\mapsto} P_2 \overset{C}{\mapsto} \dots \overset{C}{\mapsto} P_k$.

From a single encrypted predicate $\overline{p_C}$ in the above example, SP finds $\text{POP}_2^C : P_T \overset{C}{\mapsto} P_F$. With more encrypted predicates observed, SP can enhance its past result knowledge by extending POP_2^C . Before we discuss how SP extends its knowledge, we define two concepts related to a new encrypted predicate on existing partition order partitions POP_k^C for $k \geq 2$.

Definition 4.3. (Trapdoor equivalence.) Let $\overline{p_1}$ and $\overline{p_2}$ be two encrypted comparison predicates on the same attribute C of an encrypted table \overline{T} . Let $P_{ab} = \{\overline{t_i} \mid \overline{t_i} \in \overline{T} \text{ and } \Theta(\overline{p_a}, \overline{t_i}) = b\}$ for

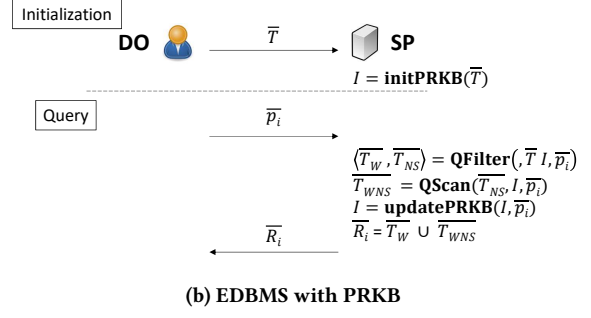
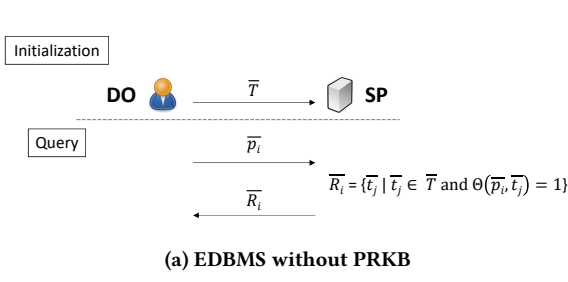


Figure 2: Communication protocol of EDBMS between DO and SP

$a = 1$ or 2 , $b = 0$ or 1 . \bar{p}_1 is said to be *equivalent* to \bar{p}_2 if either (i) $P_{10} = P_{20}$ and $P_{11} = P_{21}$; or (ii) $P_{10} = P_{21}$ and $P_{11} = P_{20}$.

Definition 4.4. (Homogeneous partition & output-isomorphic partitions.) Given $POP_k^C : P_1 \xrightarrow{C} P_2 \xrightarrow{C} \dots \xrightarrow{C} P_k$ and an encrypted predicate \bar{p} on attribute C . A partition P_a is said to be *homogeneous* w.r.t. \bar{p} if $\forall \bar{t}_i, \bar{t}_j \in P_a, \Theta(\bar{p}, \bar{t}_i) = \Theta(\bar{p}, \bar{t}_j)$. P_a is said to be *T-homogeneous* (resp. *F-homogeneous*) w.r.t. \bar{p} if $\forall \bar{t}_i \in P_a, \Theta(\bar{p}, \bar{t}_i) = 1$ (0 resp.). Two partitions P_a, P_b are said to be *output-isomorphic* w.r.t. \bar{p} if either (i) both partitions are T-homogeneous or (ii) both partitions are F-homogeneous.

We explain the intuition of the above two definitions below. Two equivalent encrypted predicates divide the encrypted table \bar{T} into the same two partitions. All encrypted tuples in a homogeneous partition have the same QPF output. A homogeneous partition is further labeled T-homogeneous (or F-homogeneous) if the QPF output is 1 (or 0) for all encrypted tuples in the partition. A non-homogeneous partition contains tuples with mixed QPF outputs, i.e., some gives 1 and some gives 0. Two output-isomorphic partitions means that all encrypted tuples in the partitions have the same QPF output. Two encrypted predicates are equivalent if they divide the encrypted tuples into the same two partitions. Note that two equivalent encrypted predicates do not necessarily mean their plain comparison predicates are the same. For example, if two comparison predicates are ' $C < 9$ ' and ' $C > 8$ ' and there is no tuple with value 8-9, the two corresponding encrypted predicates give the same two partitions, i.e., they are equivalent. Since equivalent encrypted predicates give the same partitions, only inequivalent encrypted predicates provide different partitioning information which can enhance SP's knowledge.

Now, we consider what SP observes when there is a new encrypted predicate with an existing POP_k^C . We summarize the scenario in the following lemma.

LEMMA 4.5. Given $POP_k^C : P_1 \xrightarrow{C} P_2 \xrightarrow{C} \dots \xrightarrow{C} P_k$. Let \mathbb{P} be the set of encrypted predicates for deriving POP_k^C . Let \bar{p} be a new encrypted predicate on attribute C . The following two cases must hold.

Case 1: \bar{p} is equivalent to some encrypted predicate in \mathbb{P} if and only if there is a separating point s s.t. (i) all partitions P_i for $i = 1$ to s are output-isomorphic, (ii) all partitions P_j for $j = s + 1$ to k are output-isomorphic; and (iii) P_i and P_j are not output-isomorphic for $i = 1$ to s and $j = s + 1$ to k .

Case 2: \bar{p} is inequivalent to all encrypted predicates in \mathbb{P} if and only if there is a separating point s s.t. (i) all partitions P_i for $i = 1$ to $s - 1$ are output-isomorphic, (ii) all partitions P_j for $j = s + 1$ to

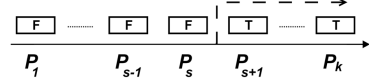


Figure 3: Example instance of Case 1 in Lemma 4.5.

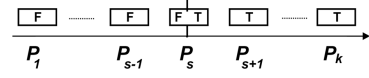


Figure 4: Example instance of Case 2 in Lemma 4.5.

k are output-isomorphic; (iii) P_i and P_j are not output-isomorphic for $i = 1$ to $s - 1$ and $j = s + 1$ to k ; and (iv) P_s is non-homogeneous.

Fig. 3 and Fig. 4 show the examples of Case 1 and Case 2.

Now, SP obtain $POP_k^C : P_1 \xrightarrow{C} P_2 \xrightarrow{C} \dots \xrightarrow{C} P_k$, generated based on a set of encrypted predicates \mathbb{P} . Note that when SP is given a new encrypted predicate \bar{p}' , SP does not know whether \bar{p}' is equivalent to some encrypted predicate in \mathbb{P} . According to Lemma 4.5, SP observes a non-homogeneous partition only in case 2. Reversely, if SP observes a non-homogeneous partition, SP can conclude that \bar{p}' is inequivalent, i.e., \bar{p}' allows SP to extend POP_k^C .

Assume now SP receives a new inequivalent encrypted predicate \bar{p}' . Let P_s be the non-homogeneous partition in POP_k^C . Since P_s is non-homogeneous, SP can divide P_s into two smaller partitions based on the outputs of Θ :

$$P_{sT} = \{\bar{t}_i \mid \bar{t}_i \in P_s \mid \Theta(\bar{p}', \bar{t}_i) = 1\} \text{ and}$$

$$P_{sF} = \{\bar{t}_i \mid \bar{t}_i \in P_s \mid \Theta(\bar{p}', \bar{t}_i) = 0\}$$

For example, in Fig. 4, the encrypted predicate divides P_s into P_{sT} on the right and P_{sF} on the left. P_{sT} and P_{sF} are now homogeneous. Either one of them must be output-isomorphic to P_{s-1} and the other partition must be output-isomorphic to P_{s+1} . Without loss of generality, assume P_{sF} is output-isomorphic to P_{s-1} and P_{sT} is output-isomorphic to P_{s+1} (like the scenario in Fig. 4). SP can conclude that $P_1 \xrightarrow{C} P_2 \xrightarrow{C} \dots \xrightarrow{C} P_{s-1} \xrightarrow{C} P_{sF} \xrightarrow{C} P_{sT} \xrightarrow{C} P_{s+1} \xrightarrow{C} P_k$. As a result, SP extends POP_k^C to POP_{k+1}^C with one more inequivalent encrypted predicate. By mathematical induction, SP can compute POP_k^C with $k - 1$ inequivalent encrypted predicates.

The above discussion only shows that SP can observe POP_k^C with $k - 1$ inequivalent encrypted predicates. We will describe how SP can efficiently update POP_k^C to POP_{k+1}^C with an additional encrypted predicate in Sec. 5.3 after we discuss how we

make use of POP_k^C to optimize selection processing of comparison predicates. POP_k^C represents the knowledge extracted from past queries. Technically, PRKB contains only one item: POP_k^C . When SP decides to build PRKB on attribute C , the algorithm $initPRKB(\bar{T})$ initiates PRKB as POP_1^C where all encrypted tuples in \bar{T} reside in one big partition. As SP receives an inequivalent encrypted predicate, SP extends its PRKB from POP_k^C to POP_{k+1}^C .

5 SINGLE COMPARISON PREDICATE PROCESSING

In this section, we describe our method of SP processing comparison predicate using PRKB. As discussed in Sec. 4, PRKB contains one single item POP_k^C , which is a set of k partitions of encrypted tuples. Let P_i be a partition in POP_k^C for $i = 1$ to k s.t. $P_1 \xrightarrow{C} P_2 \xrightarrow{C} \dots \xrightarrow{C} P_k$. Let \bar{p} be an encrypted predicate SP receives from DO. From lemma 4.5, there is a *separating point* s such that it divides the partitions (except the non-homogeneous partition in Case 2) into two groups where all partitions in the same group are output-isomorphic to each other w.r.t. \bar{p} , i.e., Θ outputs the same for all encrypted tuples in all partitions within the same group. If SP knows the value of s , SP can determine the QPF outputs of all encrypted tuples in the above two groups with 2 QPF uses only. This can significantly save the computational cost at SP. However, SP does not know s in the beginning. So, the first task of SP in processing a comparison predicate is to find out the separating point s . This is done by the algorithm $QFilter$. $QFilter$ can narrow the possible candidates of s down to just two candidates, i.e., only two out of k partitions could be non-homogeneous. Since there are always two candidates of partitions, we call them *Not-sure pair (NS-pair)*. The QPF outputs of encrypted tuples in all the other $k - 2$ partitions can be determined right away. Then, the algorithm $QScan$ will scan every encrypted tuple in NS-Pair to confirm the value of s , with early stop strategy applied.

In the following, we first present how $QFilter$ helps SP find out the separating point efficiently in Sec. 5.1. Then, we present $QScan$ in Sec. 5.2. Finally, we discuss how SP updates PRKB from POP_k^C to POP_{k+1}^C efficiently in Sec. 5.3.

5.1 QFilter: searching for NS-Pair

Before we talk about the algorithm $QFilter$, we present the following lemma about searching for separating point s .

LEMMA 5.1. *Given partial order partitions POP_k^C of an encrypted table \bar{t} and an encrypted predicate \bar{p} on C . Let P_i be a partition in POP_k^C for $i = 1$ to k s.t. $P_1 \xrightarrow{C} P_2 \xrightarrow{C} \dots \xrightarrow{C} P_k$. Let s be the separating point mentioned in lemma 4.5. Let \bar{t}_x, \bar{t}_y be 2 encrypted tuples in P_a, P_b respectively s.t. $a < b$. We have if $\Theta(\bar{p}, \bar{t}_x) = \Theta(\bar{p}, \bar{t}_y)$, then $s \leq a$ or $s \geq b$.*

In the beginning, the separating point s may be any value from 1 to k . Lemma 5.1 helps to prune the candidates of s . There are two important observations from Lemma 5.1: (i) SP just needs to test on one sample encrypted tuple in P_a and P_b ; and (ii) Lemma 5.1 does not prune the case of $s = a$ and $s = b$.

Following observation (i), SP adopts a sampling strategy to make use of the pruning shown in lemma 5.1. In the rest of the paper, we use ' $P_i.sample$ ' to denote the random encrypted tuple drawn from a partition P_i . Then, in observation (ii), since the pruning always leave at least two candidates, SP cannot confirm

the actual separating point using only sampled encrypted tuples. The sampling technique will always reduce the number of candidates to exactly 2. Thus, we call the two partitions corresponding to these 2 candidates *Not-sure pair (NS-pair)*. After SP finds NS-Pair using the sampling technique, SP scans the two partitions and confirm the separating point using $QScan$.

Algorithm 1 shows the pseudo code of $QFilter$.

Algorithm 1: QFilter

```

Input : Encrypted table  $\bar{T}$ 
Input : PRKB  $\mathcal{I} = POP_k^C : P_1 \xrightarrow{C} P_2 \xrightarrow{C} \dots \xrightarrow{C} P_k$ 
Input : Encrypted predicate  $\bar{p}_i$ 
Output :  $\langle \bar{T}_W, \bar{T}_{NS} \rangle$ 
1  $label_1 = \Theta(\bar{p}_i, P_1.sample)$ ;
2  $label_k = \Theta(\bar{p}_i, P_k.sample)$ ;
3 if  $label_1 = label_k$  then
4   // boundary case
5   if  $label_1 = 1$  then
6      $\bar{T}_W = \bigcup_{j=2}^{k-1} P_j$ ;
7   else
8      $\bar{T}_W = \phi$ ; //  $\bar{T}_W$  is empty
9   end
10   $\bar{T}_{NS} = \langle P_1, P_k \rangle$ ;
11 else
12  // use binary search to locate NS-Pair
13   $a = 1$ ; // first partition (head)
14   $b = k$ ; // last partition (tail)
15  do
16     $m = \lfloor \frac{a+b}{2} \rfloor$ ;
17     $label_m = \Theta(\bar{p}_i, P_m.sample)$ ;
18    if  $label_a = label_m$  then
19       $a = m$ ;
20    else
21       $b = m$ ;
22    end
23  while  $b - a > 1$ ;
24   $\bar{T}_{NS} = \langle P_a, P_b \rangle$ ;
25  if  $label_1 = 1$  then
26     $\bar{T}_W = \bigcup_{j=1}^{a-1} P_j$ ;
27  else
28     $\bar{T}_W = \bigcup_{j=b+1}^k P_j$ ;
29  end
30 end
31 return  $\langle \bar{T}_W, \bar{T}_{NS} \rangle$ ;

```

SP starts the search by applying Θ on $P_1.sample$ and $P_k.sample$ (line 1-2). There are two possible scenarios.

Scenario (i): if $\Theta(\bar{p}_i, P_1.sample) = \Theta(\bar{p}_i, P_k.sample)$, we call this the boundary case (line 5-10). Following Lemma 5.1, $s \leq 1$ or $s \geq k$, i.e., $s = 1$ or $s = k$. In this scenario, $\langle P_1, P_k \rangle$ is the NS-Pair returned by this phase (line 10). Any two partitions P_u and P_v for $u, v = 2$ to $k - 1$ must be output-isomorphic and have the same QPF output as the samples of P_1 and P_k . The Winner group \bar{T}_W can be found accordingly (line 5-8).

Scenario (ii): if $\Theta(\bar{p}_i, P_1.sample) \neq \Theta(\bar{p}_i, P_k.sample)$, we call this the recursive case (line 13-28). SP uses binary search to locate NS-Pair. SP applies Θ to the sample in the middle partition

$P_m.sample$ where $m = \lfloor \frac{1+k}{2} \rfloor$. If $\Theta(\overline{p}_i, P_1.sample) = \Theta(\overline{p}_i, P_m.sample)$, following lemma 5.1, $s \leq 1$ or $s \geq m$. If $s = 1$, all partitions P_i for $i > 1$ should be output-isomorphic to each other. Since $\Theta(\overline{p}_i, P_m.sample) \neq \sigma(P_k.sample)$, s cannot be 1. Thus, s must lie in $[m, k]$. SP recursively repeats the above procedure to find the separating point s in $[m, k]$. The search ends when there are two candidates left, x and $x + 1$. $\langle P_x, P_{x+1} \rangle$ is the NS-Pair returned. The rest of the partitions can be divided into two groups: the first group is P_1 to P_{x-1} and the second group is P_{x+1} to P_k . Any two partitions in the same group must be output-isomorphic and partitions in either one of the groups must be T-homogeneous. By looking at the QPF output of a sample encrypted tuple from the two groups, the Winner group \overline{T}_W can be set (line 25-28).

5.2 QScan: finding exact selection result

Let $\langle P_a, P_b \rangle$ be the NS-Pair SP obtained from *QFilter*. Encrypted tuples in these two partitions are tested using QPF Θ to see which one is actual answer in the selection result. Note that there are two cases in lemma 4.5. Case 1: \overline{p}_i is equivalent to an encrypted predicate in \mathbb{P} ; or Case 2: \overline{p}_i is inequivalent to all encrypted predicates in \mathbb{P} . The difference between the two cases is that the separating partition in Case 2 is non-homogeneous while all partitions are homogeneous in Case 1. SP can make use of the above difference and adopts an early stop strategy: SP first applies Θ on every encrypted tuples in P_a . If P_a is found to be non-homogeneous, it must be Case 2 and $s = a$. SP does not need to apply Θ on any encrypted tuples in P_b . In the other case where P_a is found to be homogeneous, SP continues to apply Θ on every encrypted tuples in P_b . If P_b is non-homogeneous, it is Case 2 and $s = b$. Otherwise, it is Case 1. In either case, *QScan* finds the set of encrypted tuples $\overline{T}_{WNS} \subseteq P_a \cup P_b$ that satisfy the predicate, i.e.,

$$\overline{T}_{WNS} = \{\overline{t}_j \mid \overline{t}_j \in P_a \cup P_b \text{ and } \Theta(\overline{p}_i, \overline{t}_j) = 1\}$$

Algorithm 2 shows the pseudo code of *QScan*.

Algorithm 2: QScan

Input : $\overline{T}_{NS} = \langle P_a, P_b \rangle$ where $a < b$
Input : PRKB $\mathcal{I} = POP_k^C : P_1 \xrightarrow{C} P_2 \xrightarrow{C} \dots \xrightarrow{C} P_k$
Input : Encrypted predicate \overline{p}_i
Output: $\overline{T}_{WNS} = \{\overline{t}_j \mid \overline{t}_j \in P_a \cup P_b \text{ and } \Theta(\overline{p}_i, \overline{t}_j) = 1\}$

- 1 // First scan P_a
- 2 $P_{a_T} = \{\overline{t}_j \mid \overline{t}_j \in P_a \text{ and } \Theta(\overline{p}_i, \overline{t}_j) = 1\}$; $P_{a_F} = P_a - P_{a_T}$;
- 3 $\overline{T}_{WNS} = P_{a_T}$;
- 4 **if** $P_{a_T} = \emptyset$ **or** $P_{a_F} = \emptyset$ **then**
- 5 // P_a is homogeneous, SP scans P_b as well
- 6 $P_{b_T} = \{\overline{t}_j \mid \overline{t}_j \in P_b \text{ and } \Theta(\overline{p}_i, \overline{t}_j) = 1\}$; $P_{b_F} = P_b - P_{b_T}$;
- 7 $\overline{T}_{WNS} = \overline{T}_{WNS} \cup P_{b_T}$;
- 8 **else**
- 9 // P_a is non-homogeneous, early stop is applied
- 10 **if** $label_b = 1$; // $label_b$ is found in *QFilter*
- 11 **then**
- 12 | $\overline{T}_{WNS} = \overline{T}_{WNS} \cup P_b$; // P_b is T-homogeneous
- 13 **end**
- 14 **end**
- 15 **return** \overline{T}_{WNS}

The complexity of the entire selection processing is $O(\frac{n}{k} + \lg n)$ where n is number of encrypted tuples in \overline{T} and k is number of partitions in PRKB.

5.3 updatePRKB: update procedure of PRKB

Recall that only inequivalent encrypted predicate can help SP to extend PRKB (see Sec. 4). During the execution of *QScan*, SP already knows whether the new encrypted predicate \overline{p}_i is equivalent to some encrypted predicate in \mathbb{P} , which generates SP's current PRKB, POP_k^C . Only when either P_a or P_b is non-homogeneous (found in *QScan*), \overline{p}_i is inequivalent (see Lemma 4.5). In such case, *QScan* (line 2 or line 6) has divided an existing partition P_s into two smaller partitions P_{s_T} and P_{s_F} where $s = a$ (line 2) or b (line 6). Without further QPF uses, SP can easily update POP_k^C to POP_{k+1}^C by replacing P_s with P_{s_T} and P_{s_F} in POP_k^C . The order of P_{s_T} and P_{s_F} in POP_{k+1}^C is determined by whether P_{s-1} is T-homogeneous or F-homogeneous. Like Fig. 4, if P_{s-1} is F-homogeneous, we have $POP_{k+1}^C : P_1 \xrightarrow{C} P_2 \xrightarrow{C} \dots P_{s-1} \xrightarrow{C} P_{s_F} \xrightarrow{C} P_{s_T} \xrightarrow{C} P_{s+1} \xrightarrow{C} \dots \xrightarrow{C} P_k$.

updatePRKB is efficient since it does not require additional QPF uses.

6 MULTI-DIMENSIONAL RANGE QUERY

In this section, we will introduce the method to optimize processing of a d -dimensional range query for $d \geq 2$. We address the most common form of d -dimensional range: the query is to retrieve all encrypted tuples with plain values falling into a d -dimensional hyper-rectangle defined by DO. The query can be described in SQL in the following form: SELECT * FROM R WHERE $c_{1a} < C_1 < c_{1b}$ AND $c_{2a} < C_2 < c_{2b}$ AND ... AND $c_{da} < C_d < c_{db}$, where C_i is an attribute in the relational table R and $c_{ia} < c_{ib}$ are query parameters defined by DO for $i = 1$ to d .

Recall that SP is not able to see the plain query parameters and the plain values of encrypted tuples. In EDBMS, the d -dimensional range query is processed as $2d$ comparison predicates (two comparisons for each dimension: $c_{ia} < C_i$ and $C_i < c_{ib}$). DO generates and gives $2d$ encrypted predicates to SP for processing the query. In existing EDBMS, SP has to apply up to $2d$ encrypted predicates on all tuples⁵, i.e., the total number of QPF uses can be $2dn$ where n is number of encrypted tuples in \overline{T} . A better alternative now is to use the single comparison predicate processing technique in Sec. 5. SP finds out the satisfying tuples of each comparison predicate. Then, SP intersects the set of satisfying tuples to find out the final selection result. Number of QPF uses can be greatly reduced compared to existing processing mechanism of EDBMS. This is our baseline method for processing multi-dimensional range query. In this section, we will describe our solution for multi-dimensional range query that is more efficient than the baseline method. In our discussion below, we will focus on 2D case for easier illustration.

6.1 Visualization of partitions on a grid

For a 2D range query, two attributes, say X and Y , are concerned. SP has maintained two partial order partitions, say $POP_{k_x}^X$ and $POP_{k_y}^Y$ of the encrypted table \overline{T} . $POP_{k_x}^X$ and $POP_{k_y}^Y$ are two partitioning ways of \overline{T} , i.e., every encrypted tuple \overline{t} in \overline{T} will be located in one and only one partition of $POP_{k_x}^X$ and one and

⁵EDBMS can stop processing for a tuple when one of the predicates is not satisfied. Actual number of QPF uses varies.

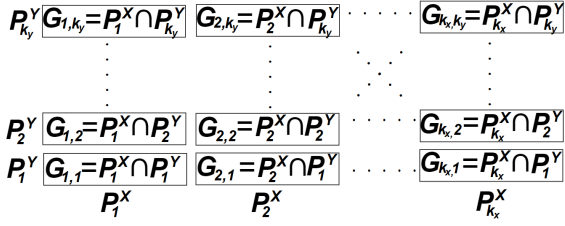


Figure 5: Visualization of the grid in 2D space

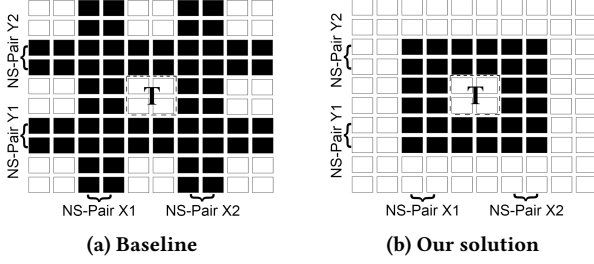


Figure 6: Illustration of partitions scanned in processing 2D range query by different methods. Encrypted tuples in the central T-region must be part of answer set.

only one partition in $POP_{k_y}^Y$. Let P_i^X be a partition in $POP_{k_x}^X$ for $i = 1$ to k_x and P_j^Y be a partition in $POP_{k_y}^Y$ for $j = 1$ to k_y . Let $G_{i,j} = P_i^X \cap P_j^Y$. We prepare a $k_x \times k_y$ grid. $G_{i,j}$ is represented as a cell at location (i, j) in the grid. Each encrypted tuple falls into one and only one grid cell $G_{i,j}$ for some i, j . The grid then represents the visualization of partitions of encrypted tuples in the 2D space. Fig. 5 shows the generated 2D grid. We remark that SP does not know the plain values of boundaries of partitions or the plain values of encrypted tuples in $G_{i,j}$, and the complete grid is not actually computed in query processing.

Now, we use the grid to visualize the processing mechanism of existing methods and identify redundancy in them. A linear scan on all encrypted tuples is equivalent to scanning all the grid cells. A better baseline solution using our single comparison predicate processing method can narrow the search on each dimension to just NS-Pairs. Only the partitions of NS-Pairs require full scan on all encrypted tuples, thus saving a significant amount of QPF uses by SP. For a 2-dimensional range query, there are two comparison predicates on each dimension. Thus, we have 4 NS-Pairs, two on each dimension to be scanned. Fig. 6a shows the illustration of scanning only the NS-Pairs in the grid. Scanning for each NS-Pair is done independently and thus a full column or row in the grid is scanned. In multi-dimensional range query, an encrypted tuple has to satisfy the comparison predicates in all dimensions. In the process of finding NS-Pairs, some of the partitions are also known to be F-homogeneous. For example, in Fig. 6a, there are 2 NS-Pairs on X. Let (P_a^X, P_{a+1}^X) and (P_b^X, P_{b+1}^X) be these two NS-Pairs, where $a < b$. Partitions from P_1^X to P_{a-1}^X must be F-homogeneous. Thus, $G_{i,j}$ for $i = 1$ to $a-1$ are not necessary to be scanned and SP can safely conclude that all encrypted tuples in these cells will not be part of the result set. Similarly, SP can apply the same pruning in other dimensions. The remaining partitions to be scanned is shown in Fig. 6b. Number of QPF uses of SP is thus reduced.

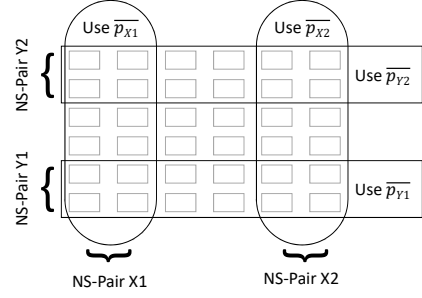


Figure 7: SP tests different encrypted predicates on encrypted tuples in different regions. NS-Pair X1, X2, Y1 and Y2 are generated according to encrypted predicates $\overline{p_{X1}}$, $\overline{p_{X2}}$, $\overline{p_{Y1}}$ and $\overline{p_{Y2}}$ respectively

6.2 Systematic scanning procedure for multi-dimensional range query

In this section, we present how SP can systematically and efficiently perform the scan. There are two major issues we need to address to achieve efficiency.

First, note that SP does not test all the encrypted predicates on the encrypted tuples in the area shown in Fig. 6b. Recall that there are $2d$ encrypted predicates where d is the number of dimension of the query. Each encrypted predicate gives an NS-Pair. Let \overline{p} be the encrypted predicate giving the NS-Pair (P_a, P_b) . Only grid cells that are computed by intersecting P_a or P_b with other partitions require testing by \overline{p} . For example, Fig. 7 shows the encrypted predicates needed for different cells in Fig. 6b.

Second, as we presented in Sec. 5.2, an early stop strategy can be used to further reduce the number of QPF uses by SP in comparison predicate processing. SP can use the same strategy in processing multi-dimensional range query as well. Each dimension has two comparison predicates resulting in two NS-Pairs. Let (P_a, P_{a+1}) and (P_b, P_{b+1}) be the two NS-Pairs, where $a < b$. We call P_{a+1} and P_b the *inner NS-partition*; and P_a and P_{b+1} the *outer NS-partition*. If SP first scans the outer NS-partition and finds that it is non-homogeneous, SP can further conclude that the inner NS-partition is T-homogeneous. On the other hand, if SP scans the inner NS-partition first and finds out that it is non-homogeneous, SP can conclude that the outer NS-partition of the same NS-Pair is F-homogeneous. Besides, once an encrypted tuple is found to have QPF output 0 for an encrypted predicate, it can never be in the selection result. SP does not need to test other encrypted predicates on this encrypted tuple.

In summary, the procedure to process multi-dimensional range query is done by the following steps:

- (1) Use *QFilter* to find the NS-Pair for each encrypted predicate.
- (2) Compute the required grid cells, e.g., in Fig. 6b, by intersecting the partitions in different $POP_{k_i}^{C_i}$ for different attributes C_i .
- (3) Test encrypted predicates using QPF on encrypted tuples in different regions of the grid, e.g., according to Fig. 7, and apply early stop strategy when possible. (Details are described above in this section.)
- (4) Return as selection result those encrypted tuples with QPF output 1 for all encrypted predicates in step (3) and the encrypted tuples in the central T-region, e.g., in Fig. 6b.

The entire process takes $O(d(\frac{n\alpha^{d-1}}{k} + \lg k))$ where n is number of encrypted tuples in \overline{T} , k is number of partitions in PRKB, d is

number of dimensions of the query and α is the selectivity on each dimension. Assume α remains the same, the query cost decreases as d increases. Our selection processing technique for multi-dimensional range query is scalable to number of dimensions.

7 DATABASE UPDATE HANDLING

The selection processing techniques we presented in Sec. 5 and Sec. 6 are designed for a static database where the contents of encrypted tuples do not change. In this section, we discuss how we can support update operations in a database.

There are 3 kinds of update operations in SQL: (1) INSERT statements; (2) DELETE statements; and (3) UPDATE statements. UPDATE statements can be considered as insertion of a new tuple after deletion of an existing tuple. We just need to cater for insertion and deletion.

7.1 Insertion Handling

When there is a new encrypted tuple to be inserted to EDBMS, SP needs to update PRKB, POP_k^C , to assign the new encrypted tuple to the correct partition. To facilitate the update, SP needs to remember the set of past $k - 1$ encrypted predicates \mathbb{P} that generates POP_k^C . SP can order the encrypted predicates in \mathbb{P} according to $POP_k^C: P_1 \xrightarrow{C} P_2 \xrightarrow{C} \dots \xrightarrow{C} P_k$ because the encrypted predicates are the separators that form the partitions. Let \bar{p}_x be the encrypted predicate s.t. partitions P_i for $i = 1$ to x are output-isomorphic and partitions P_j for $j = x + 1$ to k are output-isomorphic but P_i and P_j are not output-isomorphic, e.g., the encrypted predicate in Fig. 3 is refereed as \bar{p}_3 . A binary search can be used by SP to find out the partition the new encrypted tuple belongs to: SP first uses the encrypted predicate in the middle \bar{p}_m , where $m = \lfloor \frac{k}{2} \rfloor$, to determine whether the encrypted tuple belongs to the first half or the second half of POP_k^C ; then recursively reduces the list by half until only one partition remains and this partition is where the new encrypted tuple belongs to.

It takes $O(\lg k)$ time to update PRKB for the new encrypted tuple. Let β be number of attributes with indexing. The total update cost is then $O(\beta \lg k)$.

7.2 Deletion Handling

Deletion handling is easy as SP simply removes the corresponding encrypted tuple from the corresponding partitions. When there is no tuple remained in a partition of POP_k^C , the partition is removed from POP_k^C , i.e., the knowledge of partial order partitions becomes POP_{k-1}^C .

8 EMPIRICAL STUDIES

There are two purposes in our experiments. First, as we mentioned in Sec. 3.3, SP can observe partial order information in existing EDBMS (even without implementing PRKB). We want to evaluate whether existing EDBMS model is acceptable in practice. Second, we empirically evaluate the performance of our indexing method, PRKB.

The experiment settings and its result for the first purpose is presented in Sec. 8.1. Information of experiments for the second purpose is presented in Sec. 8.2.

		Number of queries				
Victims	Size	250	1K	10K	100K	1M
Hospital	2,426,516	0.007	0.020	0.115	0.605	2.846
Labor	6,156,470	0.042	0.117	0.484	1.673	5.807
Latitude	1,122,932	0.008	0.025	0.212	1.650	11.167
Longitude	1,122,932	0.011	0.038	0.331	2.440	13.592

Table 2: Recovered portion of ordering information (RPOI) (%) on real datasets varying number of queries observed by attacker

8.1 Experiment on security of EDBMS model revealing selection result

As discussed in Sec. 3.3, we want to see how much partial order information can be derived by SP/attacker in existing EDBMS model in practice. To quantify how close the recovered partial order is to total order, we define *recovered portion of ordering information* (RPOI) as $\frac{\text{Recovered partial order length}}{\text{Total order length}}$. (Partial order length is the size of the longest chain, e.g., the length of total order is n for a dataset of n distinct numbers.)

We follow the scenario used in [24] to perform the experiment. SP is able to receive certain number of queries, randomly generated by DO. Unlike [24], SP in our case receives limited number of queries only and we pick attributes with large domain sizes as victims. Each query has has one encrypted predicate. We vary number of queries from 250 to 1M and measure RPOI in these cases. We tested on 4 victim attributes from 3 different real datasets:

- (1) **Hospital Charges:** Hospital Inpatient Discharges 2013 dataset⁶
- (2) **Labor Salary:** US Labor Statistic 2017⁷
- (3) **Latitude:** US Buildings dataset⁸
- (4) **Longitude:** US Buildings dataset

The result is presented in Table 2.

The result shows that the partial order information observed by SP is still far from complete as the total order. RPOI increases at decreasing speed as SP observes more queries. It is because it gets harder for SP to observe a useful query to enhance the partial order knowledge. According to Quantcast⁹, it could take weeks for a top-1000 website to get a million of traffic, which is still far away from recovering the total order in an attack attempt. We consider our current model of EDBMS revealing selection result as practically secure for large domain data. In contrast, if OPE is used, e.g., in CryptDB [29], RPOI is 100% even SP has not yet processed any query.

8.2 Performance evaluation of PRKB

8.2.1 Algorithm implementation. We separately evaluate single-dimensional (SD) query and multi-dimensional (MD) range query. Note that there are different processing techniques in using PRKB. To differentiate them, we use (i) ‘PRKB(SD)’ to denote the processing method for single-dimensional query (Sec. 5), (ii) ‘PRKB(SD+)’ to denote the naive extension of PRKB(SD) for multi-dimensional range query (see Sec. 6), and (iii) ‘PRKB(MD)’ to denote the algorithm designed for multi-dimensional range query (see Sec. 6.2).

⁶<https://health.data.ny.gov/>

⁷<https://catalog.data.gov/>

⁸<http://www.geonames.org/>

⁹<https://www.quantcast.com/top-sites>

As a competitor, we implemented the indexing method ‘Logarithmic-SRC-i’ in [12]. Note that Logarithmic-SRC-i may return false positives to DO. DO needs to decrypt them to confirm whether they are actual answer in the selection result. This may require a significant amount of DO’s involvement to process the query. In our implementation, we deployed a trusted machine (TM), like Cipherbase [2], to perform this confirmation process on behalf of DO. In PRKB, we use the same confirmation process as QPF, i.e., SP sends the encrypted data to TM; TM decrypts and returns the QPF output of the encrypted tuple. Besides, Logarithmic-SRC-i is an encrypted index computed and maintained by DO. This is also done by TM in our implementation. In our experiment, both TM and SP equip with a machine with the same power. We also compare with the case where no indexing is used, denoted as Baseline.

PRKB replies on past queries to operate effectively. In all experiments, number of queries is limited to small values (at most 600) so as to show that PRKB is effective even without a lot of past result knowledge.

All algorithms were implemented in C/C++. All machines in the experiment equip with 2GHz CPU and 4GB RAM running Linux platform.

8.2.2 Datasets and Tests. We performed our experiments on both synthetic and real datasets. We performed most of the experiments on synthetic datasets to evaluate the performance varying different parameters, e.g., number of tuples and selectivity. In the synthetic datasets, the data domain of all attributes is set to be integers in $[1, 30M]$. The plain value on each attribute of each tuple is randomly generated¹⁰.

We simulated a use case on a real dataset. The US buildings dataset contains 1,122,932 records about information of buildings in US, including location (latitude and longitude). A tourist (user) is interested to know what buildings are around the location he will visit. The user issues a range query to retrieve all buildings within a $1km \times 1km$ region in the dataset, i.e., it is a 2D range query.

We measure the average number of QPF uses¹¹ (# QPF use) and average execution time out of 20 runs for each experiment.

8.2.3 Experiment on Building PRKB. This experiment simulates SP building PRKB from scratch on a synthetic dataset with 10M tuples. We assume SP receives 600 distinct queries, each with one comparison predicate, and we monitor the performance of query processing cost. For reference, the performance of Baseline and Logarithmic-SRC-i is also shown. Fig. 8 shows the result of query cost and Table 3 shows the space consumption of PRKB.

We make the following observations from the result:

- (1) In the beginning, PRKB has no knowledge. Query processing is as slow as Baseline. However, when SP receives queries, the cost drops fast. At 50-th query, the cost has already dropped by an order of magnitude and PRKB has almost the same performance as Logarithmic-SRC-i. At 600-th query, the query time of PRKB is one order of magnitude smaller than Logarithmic-SRC-i. It shows that PRKB is practical, reducing the query processing cost significantly with a small amount of past result knowledge.

¹⁰We have tested on data generated with different distributions, including uniform, normal, correlated and anti-correlated. The results are similar and so we just present the results for uniform distribution in this paper.

¹¹Since majority of actions in Logarithmic-SRC-i are related to its index structure, we do not show # QPF use for Logarithmic-SRC-i.

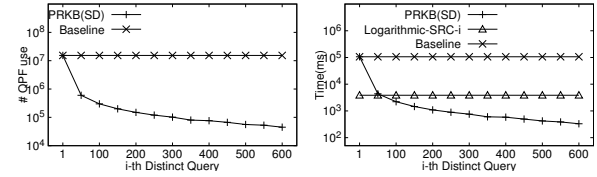


Figure 8: Performance of Query with growing PRKB on 10M tuples (1% Selectivity)

Method	Dataset size (in millions)					
	10	12	14	16	18	20
PRKB-250	38.2	45.8	53.4	61.0	68.7	76.3
PRKB-600	38.2	45.9	53.5	61.1	68.8	76.4
Logarithmic-SRC-i	3589	4050	4493	4918	6356	6758

Table 3: Storage size of the index (in MB)

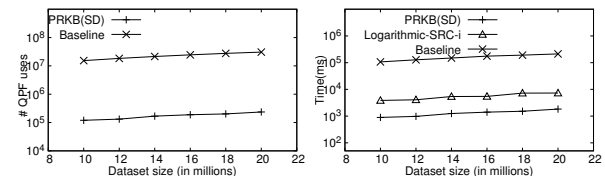


Figure 9: Performance on single-dimensional query varying dataset size (1% selectivity)

- (2) PRKB occupies a small space, as PRKB is simply partition information of encrypted tuples. There is a slight increase in space consumption (from 76.3MB to 76.4MB for 20M dataset) of PRKB. It is because SP needs to keep more encrypted predicates to handle database update (see Sec. 7). The increase in space consumption is negligible. Logarithmic-SRC-i requires much more space due to its more complex index structure.
- (3) The query processing cost is consistent with number of QPF uses. This shows that QPF computation is the dominant cost in EDBMS. Reducing number of QPF uses can help to reduce the overall query cost.

8.2.4 Experiment for Single-dimensional Query. We tested the performance of algorithms in handling a single-dimensional query under different settings. The query is in the form of “SELECT * FROM Dataset WHERE $lb < X < ub$ ”. X is an attribute on synthetic dataset. lb and ub are two parameters generated randomly according to selectivity. We use a static PRKB with 250 partitions for the experiment. There are 2 parameters in the experiment: (i) dataset size, varying from 10M to 20M tuples; (ii) selectivity, varying from 1% to 10%.

Fig. 9 and Fig. 10 show the results of experiments in varying dataset size and selectivity respectively.

We make the following observations from the results.

- (1) All algorithms scale well with increasing number of tuples. Cost reduction of PRKB(SD) over Baseline and Logarithmic-SRC-i is consistent, at about two orders of magnitude and a factor of 4, respectively.
- (2) PRKB(SD) shows a steady performance no matter how selectivity increases. It is because PRKB simply requires SP to examine two NS-Pairs defining the boundary of the

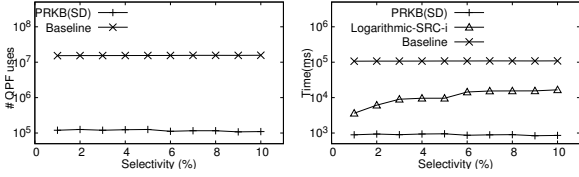


Figure 10: Performance on single-dimensional query varying selectivity (dataset of 10M tuples)

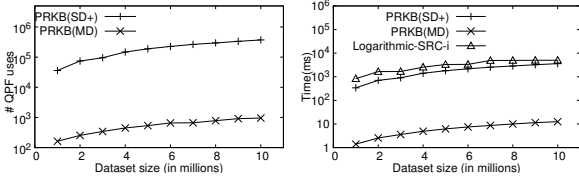


Figure 11: Performance on multi-dimensional query varying dataset size (Dimensionality of 3, 2% selectivity per dimension)

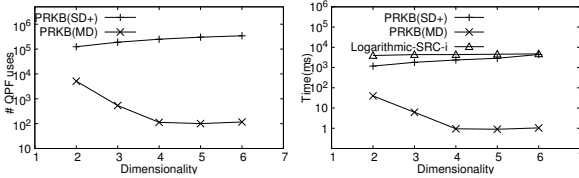


Figure 12: Performance on multi-dimensional query varying dimensionality (Dataset of 5M tuples, 2% selectivity per dimension)

answer set. All encrypted tuples in partitions between the two NS-Pairs can be returned as answer without applying QPF on them. The cost of PRKB is independent to size of answer set.

8.2.5 Experiment for Multi-dimensional Range Query. In this experiment, we study the difference in performance between PRKB(SD+), PRKB(MD) and Logarithmic-SRC-i under different settings to validate the importance of our optimization method for handling multi-dimensional range query. The range query tested is in the form of “SELECT * FROM Dataset WHERE $lb_1 < X_1 < ub_1$ AND ... AND $lb_d < X_d < ub_d$ ”. X_i is an attribute in the synthetic dataset. lb_i and ub_i are generated randomly according to selectivity (per dimension), which is set to be 2%. Both algorithms use a static PRKB with 250 partitions. There are 2 parameters in the experiment: (i) dimensionality d , varying from 2 to 6, and (ii) dataset size, varying from 1M to 10M tuples.

Figure 11 and 12 show the results. Improvement of PRKB(MD) over PRKB(SD+) and Logarithmic-SRC-i is consistent with increasing dataset size. The cost of PRKB(SD+) increases as number of dimensions increases because PRKB(SD+) processes each dimension separately. However, number of results actually decreases with more comparison predicates. Logarithmic-SRC-i sent a set of hashed values for keyword search for each dimensions. The cost of Logarithmic-SRC-i is getting closer to PRKB(SD+) in Figure 12. PRKB(MD), on the other hand, can make use of the fact that more comparison predicates filter more candidate tuples. Thus, the cost of PRKB(MD) decreases with

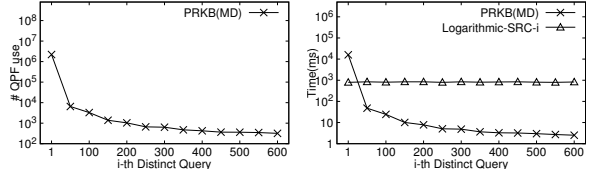


Figure 13: Performance of Query with growing PRKB on US buildings dataset (2% selectivity)

Method	Batch				
	1	2	3	4	5
PRKB	32,356	32,104	32,117	32,167	32,168
Logarithmic-SRC-i	2,936	2,967	2,967	2,935	2,937

Table 4: Average throughput (Tuples / Second) of inserting 5 batches (each with 2M tuples) of data to PRKB with 10M tuples

increasing number of dimensions. PRKB(MD) can perform well even for higher dimensional range queries.

8.2.6 Experiment on Real Dataset. We tested PRKB and Logarithmic-SRC-i in a simulated use case (described in Sec. 8.2.2) on real dataset to validate its practicality.

In this dataset, the space consumed by PRKB is less than 1% of the size of encrypted dataset ($\frac{8.81MB}{1.04GB}$) while Logarithmic-SRC-i consumed more than 43% space ($\frac{441.346MB}{1.04GB}$).

Similar to the experiments on synthetic datasets, the query processing time is high in the beginning. Initially, the query time of Logarithmic-SRC-i is smaller than that of PRKB. After answering 50 queries, the query time of PRKB is already below 100ms and performs better than Logarithmic-SRC-i. After answering 600 queries, the query time of PRKB is further reduced to 9ms. In contrast, if EBDMS does not use any index, it takes 15.9s to process a query, which is impractical in reality. Besides, if DO wants to avoid the poor performance of EBDMS using PRKB in the beginning, DO can arbitrarily generates queries (as few as 50 queries in this case) to help SP to build an initiate PRKB.

8.2.7 Experiment for Handling Database Update. In this experiment, we evaluate the cost of SP updating PRKB in handling database update. Since deletion is simple, we only show the results for insertion here. PRKB has 250 partitions. The experiment is done on a synthetic dataset with 10M tuples. We inserted 5 batches, each with 2M new tuples, to the database, i.e., the database contains 20M tuples in the end. We measure the average throughput (number of tuples inserted per second) achieved by PRKB in each batch. For comparison, we measure the throughput of Logarithmic-SRC-i in the same setting. Table 4 shows the result.

The throughput of PRKB remains almost the same. The observation can be explained in our analysis in Sec. 7.1, as the update cost is independent to database size. SP can easily bear the update cost to maintain PRKB for optimizing query processing.

9 CONCLUSIONS AND FUTURE WORK

In this paper, we proposed a novel indexing method - past result knowledge base (PRKB) for EDBMS. Unlike traditional indexing problem for encrypted data, PRKB is built solely by SP based on results of past queries. None of existing indexing methods work without DO’s involvement or customized encryption method.

We showed that PRKB is effective in reducing the processing cost of new queries. Our experiments showed that PRKB consistently outperforms a state-of-the-art competitor in [12] and PRKB achieves a speed-up of at least an order of magnitude compared to EDBMS without implementing PRKB. Since SP is just making use the information that is already available to SP to build PRKB, security of PRKB is ensured. In the future, we plan to extend PRKB to incorporate different query result and to support more query types. The partial order information in PRKB can also be used in optimizing queries like Min, Max or Skyline queries.

ACKNOWLEDGEMENTS

The research is supported by FDS grant (UGC/FDS14/E05/14).

A SUPPORTING BETWEEN OPERATOR

Some methods, e.g., [12], support specifically BETWEEN operator. BETWEEN operator returns the overall result of whether the encrypted tuple falls into the range instead of two results of two comparisons, i.e., SP observes less information for BETWEEN. In fact, as we will show below, BETWEEN is equivalent to two separate comparisons, w.r.t. building PRKB, in most cases.

Say SP has $POP_5^C : P_1 \xrightarrow{C} P_2 \xrightarrow{C} P_3 \xrightarrow{C} P_4 \xrightarrow{C} P_5$. Let \bar{p} be an encrypted predicate that computes 'X BETWEEN a and b'. We can derive a similar observation like lemma 4.5 that, in general, Θ returns 1 for encrypted tuples in partitions in the middle and 0 for encrypted tuples in partitions in the two ends. Say (i) P_3 is T-homogeneous, (ii) P_1, P_5 are F-homogenous, and (iii) P_2 and P_4 are non-homogeneous. Each of P_2 and P_4 is divided into two partitions and we have P_{2T}, P_{2F}, P_{4T} , and P_{4F} where P_{iT} (P_{iF} resp.) denotes the set of tuples in P_i that get 1 (0 resp.) from Θ . SP obtains $POP_7^C : P_1 \xrightarrow{C} P_{2F} \xrightarrow{C} P_{2T} \xrightarrow{C} P_3 \xrightarrow{C} P_{4T} \xrightarrow{C} P_{4F} \xrightarrow{C} P_5$. SP obtains the same POP_7^C as if SP obtains two encrypted predicates for ' $X \geq a$ ' and ' $X \leq b$ '. The BETWEEN predicate reveals the same partial order information to SP as two separate comparison predicates in this scenario.

Only when the range in the BETWEEN operator is very small such that only some encrypted tuples in one partition get 1 from Θ , SP cannot determine the order information of other tuples in this partition.

Computing a BETWEEN operator using PRKB is similar to comparison handling. SP looks for two separating points using the samples of partitions, like *QFilter*. When a sample encrypted tuple with QPF output 1 is found, two binary searches are performed to find two NS-pairs, each containing a separating point on the two ends of the range of BETWEEN predicate. The process after that is the same as comparison handling. However, when no sample with QPF output 1 is found, SP cannot conclude whether other tuples will return 1 or 0 from QPF due to the existence of the above exceptional case. SP needs to draw more samples from partitions. The worst case is that SP finds that there is no satisfying tuple after scanning all encrypted tuples.

REFERENCES

- [1] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. 2004. Order-Preserving Encryption for Numeric Data. In *SIGMOD*.
- [2] Arvind Arasu, Spyros Blanas, Ken Eguro, Manas Joglekar, Raghav Kaushik, Donald Kossmann, Ravishankar Ramamurthy, Prasang Upadhyaya, and Ramarathnam Venkatesan. 2013. Secure database-as-a-service with Cipherbase. In *SIGMOD*.
- [3] Arvind Arasu, Spyros Blanas, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravi Ramamurthy, and Ramarathnam Venkatesan. 2013. Orthogonal Security With Cipherbase. In *CIDR*.

- [4] Arvind Arasu, Ken Eguro, Manas Joglekar, Raghav Kaushik, Donald Kossmann, and Ravi Ramamurthy. 2015. Transaction processing on confidential data using cipherbase. In *ICDE*.
- [5] Sumeet Bajaj and Radu Sion. 2011. TrustedDB: a trusted hardware based database with privacy and data confidentiality. In *SIGMOD*.
- [6] Alexandra Boldyreva, Nathan Chenette, and Adam O'Neill. 2011. Order-Preserving Encryption Revisited: Improved Security Analysis and Alternative Solutions. In *CRYPTO*.
- [7] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. 2014. Dynamic Searchable Encryption in Very-Large Databases: Data Structures and Implementation. In *NDSS*.
- [8] David Cash, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. 2013. Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries. In *CRYPTO*.
- [9] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. 1998. Private Information Retrieval. *JACM* 45, 6 (1998).
- [10] Reza Curtmola, Juan A. Garay, Seny Kamara, and Rafail Ostrovsky. 2006. Searchable symmetric encryption: improved definitions and efficient constructions. In *CCS*.
- [11] Ernesto Damiani, Sabrina De Capitani di Vimercati, Sushil Jajodia, Stefano Paraboschi, and Pierangela Samarati. 2003. Balancing confidentiality and efficiency in untrusted relational DBMSs. In *CCS*.
- [12] Ioannis Demertzis, Stavros Papadopoulos, Odysseas Papapetrou, Antonios Deligiannakis, and Minos N. Garofalakis. 2016. Practical Private Range Search Revisited. In *SIGMOD*.
- [13] Ioannis Demertzis and Charalampos Papamanthou. 2017. Fast Searchable Encryption with Optimal Locality. In *SIGMOD*.
- [14] Sabrina De Capitani di Vimercati, Sara Foresti, Stefano Paraboschi, Gerardo Pelosi, and Pierangela Samarati. 2015. Shuffle Index: Efficient and Private Access to Outsourced Data. *TOS* 11, 4 (2015).
- [15] O. Goldreich, S. Micali, and A. Wigderson. 1987. How to Play any Mental Game. In *STOC*.
- [16] Oded Goldreich and Rafail Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *J. ACM* 43, 3 (1996).
- [17] Chunsheng Gu and Jixing Gu. 2014. Known-plaintext attack on secure kNN computation on encrypted databases. *Security and Communication Networks* 7, 12 (2014).
- [18] Hakan Hacigümüs, Balakrishna R. Iyer, Chen Li, and Sharad Mehrotra. 2002. Executing SQL over encrypted data in the database-service-provider model. In *SIGMOD*.
- [19] Zhian He, Wai Kit Wong, Ben Kao, David Wai-Lok Cheung, Rongbin Li, Siu-Ming Yiu, and Eric Lo. 2015. SDB: A Secure Query Processing System with Data Interoperability. *PVLDB* 8, 12 (2015).
- [20] Bijit Hore, Sharad Mehrotra, Mustafa Canim, and Murat Kantarcioglu. 2012. Secure Multidimensional Range Queries over Outsourced Data. *The VLDB Journal* (2012).
- [21] Bijit Hore, Sharad Mehrotra, and Gene Tsudik. 2004. A Privacy-Preserving Index for Range Queries. In *VLDB*.
- [22] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2014. Inference attack against encrypted range queries on outsourced databases. In *CODASPY*.
- [23] Panagiotis Karras, Artyom Nikitin, Muhammad Saad, Rudrika Bhatt, Denis Antyukhov, and Stratos Idreos. 2016. Adaptive Indexing over Encrypted Numeric Data. In *SIGMOD*.
- [24] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. 2016. Generic Attacks on Secure Outsourced Databases. In *CCS*.
- [25] Rui Li, Alex X. Liu, Ann L. Wang, and Bezawada Bruhadeshwar. 2014. Fast Range Query Processing with Strong Privacy Protection for Cloud Computing. *PVLDB* 7, 14 (2014).
- [26] Sha Ma, Bo Yang, Kangshun Li, and Feng Xia. 2011. A Privacy-Preserving Join on Outsourced Database. In *ISC*.
- [27] Muhammad Naveed, Seny Kamara, and Charles V. Wright. 2015. Inference Attacks on Property-Preserving Encrypted Databases. In *SIGSAC*.
- [28] Raluca A. Popa, Frank H. Li, and Nikolai Zeldovich. 2013. An Ideal-Security Protocol for Order-Preserving Encoding. In *SP*.
- [29] Raluca A. Popa, Catherine M. S. Redfield, Nikolai Zeldovich, and Hari Balakrishnan. 2011. CryptDB: protecting confidentiality with encrypted query processing. In *SOSP*.
- [30] Elaine Shi, John Bethencourt, T-H. Hubert Chan, Dawn Song, and Adrian Perig. 2007. Multi-Dimensional Range Query over Encrypted Data. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy (SP '07)*.
- [31] Erez Shmueli, Ronen Waisenberg, Yuval Elovici, and Ehud Gudes. 2005. Designing Secure Indexes for Encrypted Databases. In *DBSec*.
- [32] E. Stefanov and E. Shi. 2013. ObliviStore: High Performance Oblivious Cloud Storage. In *SP*.
- [33] Stephen Tu, M. Frans Kaashoek, Samuel Madden, and Nikolai Zeldovich. 2013. Processing Analytical Queries over Encrypted Data. *PVLDB* 6, 5 (2013).
- [34] Wai Kit Wong, David Wai-Lok Cheung, Ben Kao, and Nikos Mamoulis. 2009. Secure kNN computation on encrypted databases. In *SIGMOD*.
- [35] Wai Kit Wong, Ben Kao, David Wai-Lok Cheung, Rongbin Li, and Siu-Ming Yiu. 2014. Secure query processing with data interoperability in a cloud database environment. In *SIGMOD*.

Temporally-Biased Sampling for Online Model Management

Brian Hentschel*
Harvard University
bhentschel@g.harvard.edu

Peter J. Haas*
University of Massachusetts
phaas@cs.umass.edu

Yuanyuan Tian
IBM Research – Almaden
ytian@us.ibm.com

ABSTRACT

To maintain the accuracy of supervised learning models in the presence of evolving data streams, we provide temporally-biased sampling schemes that weight recent data most heavily, with inclusion probabilities for a given data item decaying exponentially over time. We then periodically retrain the models on the current sample. This approach speeds up the training process relative to training on all of the data. Moreover, time-biasing lets the models adapt to recent changes in the data while—unlike in a sliding-window approach—still keeping some old data to ensure robustness in the face of temporary fluctuations and periodicities in the data values. In addition, the sampling-based approach allows existing analytic algorithms for static data to be applied to dynamic streaming data essentially without change. We provide and analyze both a simple sampling scheme (T-TBS) that probabilistically maintains a target sample size and a novel reservoir-based scheme (R-TBS) that is the first to provide both complete control over the decay rate and a guaranteed upper bound on the sample size, while maximizing both expected sample size and sample-size stability. The latter scheme rests on the notion of a “fractional sample” and, unlike T-TBS, allows for data arrival rates that are unknown and time varying. R-TBS and T-TBS are of independent interest, extending the known set of unequal-probability sampling schemes. We discuss distributed implementation strategies; experiments in Spark illuminate the performance and scalability of the algorithms, and show that our approach can increase machine learning robustness in the face of evolving data.

1 INTRODUCTION

A key challenge for machine learning (ML) is to keep ML models from becoming stale in the presence of evolving data. In the context of the emerging Internet of Things (IoT), for example, the data comprises dynamically changing sensor streams [26], and a failure to adapt to changing data can lead to a loss of predictive power.

One way to deal with this problem is to re-engineer existing static supervised learning algorithms to become adaptive. Some parametric algorithms such as SVM can indeed be re-engineered so that the parameters are time-varying, but for non-parametric algorithms such as kNN-based classification, it is not at all clear how re-engineering can be accomplished. We therefore consider alternative approaches in which we periodically retrain ML models, allowing static ML algorithms to be used in dynamic settings essentially as-is. There are several possible retraining approaches.

Retraining on cumulative data: Periodically retraining a model on all of the data that has arrived so far is clearly infeasible because of the huge volume of data involved. Moreover, recent

data is swamped by the massive amount of past data, so the retrained model is not sufficiently adaptive.

Sliding windows: A simple sliding-window approach would be to, e.g., periodically retrain on the data from the last two hours. If the data arrival rate is high and there is no bound on memory, then one must deal with long retraining times caused by large amounts of data in the window. The simplest way to bound the window size is to retain the last n items. Alternatively, one could try to subsample within the time-based window [14]. The fundamental problem with all of these bounding approaches is that old data is completely forgotten; the problem is especially severe when the data arrival rate is high. This can undermine the robustness of an ML model in situations where old patterns can reassert themselves. For example, a singular event such as a holiday, stock market drop, or terrorist attack can temporarily disrupt normal data patterns, which will reestablish themselves once the effect of the event dies down. Periodic data patterns can lead to the same phenomenon. Another example, from [27], concerns influencers on Twitter: a prolific tweeter might temporarily stop tweeting due to travel, illness, or some other reason, and hence be completely forgotten in a sliding-window approach. Indeed, in real-world Twitter data, almost a quarter of top influencers were of this type, and were missed by a sliding window approach.

Temporally biased sampling: An appealing alternative is a temporally biased sampling-based approach, i.e., maintaining a sample that heavily emphasizes recent data but also contains a small amount of older data, and periodically retraining a model on the sample. By using a time-biased sample, the retraining costs can be held to an acceptable level while not sacrificing robustness in the presence of recurrent patterns. This approach was proposed in [27] in the setting of graph analysis algorithms, and has recently been adopted in the MacroBase system [3]. The orthogonal problem of choosing when to retrain a model is also an important question, and is related to, e.g., the literature on “concept drift” [13]; in this paper we focus on the problem of how to efficiently maintain a time-biased sample.

In more detail, our time-biased sampling algorithms ensure that the “appearance probability” for a given data item—i.e., the probability that the item appears in the current sample—decays over time at a controlled exponential rate. Specifically, we assume that items arrive in batches (see the next section for more details), and our goal is to ensure that (i) our sample is representative in that all items in a given batch are equally likely to be in the sample, and (ii) if items i and j belong to batches that have arrived at (wall clock) times t' and t'' with $t' \leq t''$, then for any time $t \geq t''$ our sample S_t is such that

$$\Pr[i \in S_t] / \Pr[j \in S_t] = e^{-\lambda(t''-t')}. \quad (1)$$

Thus items with a given timestamp are sampled uniformly, and items with different timestamps are handled in a carefully controlled manner. The criterion in (1) is natural and appealing in applications and, importantly, is interpretable and understandable to users. As discussed in [27], the value of the decay rate λ can be chosen to meet application-specific criteria. For example, by setting $\lambda = 0.058$, around 10% of the data items from 40 batches

*Work performed at IBM Research – Almaden

ago are included in the current analysis. As another example, suppose that, $k = 150$ batches ago, an entity such as a person or city was represented by $n = 1000$ data items and we want to ensure that, with probability $q = 0.01$, at least one of these data items remains in the current sample. Then we would set $\lambda = -k^{-1} \ln(1 - (1 - q)^{1/n}) \approx 0.077$. If training data is available, λ can also be chosen to maximize accuracy via cross validation.

The exponential form of the decay function has been adopted by the majority of time-biased-sampling applications in practice because otherwise one would typically need to track the arrival time of every data item—both in and outside of the sample—and decay each item individually at an update, which would make the sampling operation intolerably slow. (A “forward decay” approach that avoids this difficulty, but with its own costs, has been proposed in [9]; we plan to investigate forward decay in future work.) Exponential decay functions make update operations fast and simple.

For the case in which the item-arrival rate is high, the main issue is to keep the sample size from becoming too large. On the other hand, when the incoming batches become very small or widely spaced, the sample sizes for all of the time-biased algorithms that we discuss (as well as for sliding-window schemes based on wall-clock time) can become small. This is a natural consequence of treating recent items as more important, and is characteristic of any sampling scheme that satisfies (1). We emphasize that—as shown in our experiments—a smaller, but carefully time-biased sample typically yields greater prediction accuracy than a sample that is larger due to overloading with too much recent data or too much old data. I.e., more sample data is not always better. Indeed, with respect to model management, this decay property can be viewed as a feature in that, if the data stream dries up and the sample decays to a very small size, then this is a signal that there is not enough new data to reliably retrain the model, and that the current version should be kept for now.

It is surprisingly hard to both enforce (1) and to bound the sample size. As discussed in detail in Section 7, prior algorithms that bound the sample size either cannot consistently enforce (1) or cannot handle wall-clock time. Examples of the former include algorithms based on the A-Res scheme of Efraimidis and Spirakis [12], and Chao’s algorithm [5]. A-Res enforces conditions on the *acceptance* probabilities of items; this leads to appearance probabilities which, unlike (1), are both hard to compute and not intuitive. A similar example is provided by Chao’s algorithm [5]. In Appendix D of [16] we demonstrate how the algorithm can be specialized to the case of exponential decay and modified to handle batch arrivals. We then show that the resulting algorithm fails to enforce (1) either when initially filling up an empty sample or in the presence of data that arrives slowly relative to the decay rate, and hence fails if the data rate fluctuates too much. The second type of algorithm, due to Aggarwal [1] can only control appearance probabilities based on the indices of the data items. For example, after n items arrive, one could require that, with 95% probability, the $(n-k)$ th item should still be in the sample for some specified $k < n$. If the data arrival rate is constant, then this might correspond to a constraint of the form “with 95% probability a data item that arrived 10 hours ago is still in the sample”, which is often more natural in applications. For varying arrival rates, however, it is impossible to enforce the latter type of constraint, and a large batch of arriving data can prematurely flush out older data. Thus our new sampling schemes are interesting in their

own right, significantly expanding the set of unequal-probability sampling techniques.

T-TBS: We first provide and analyze Targeted-Size Time-Biased Sampling (T-TBS), a simple algorithm that generalizes the sampling scheme in [27]. T-TBS allows complete control over the decay rate (expressed in wall-clock time) and probabilistically maintains a target sample size. That is, the expected and average sample sizes converge to the target and the probability of large deviations from the target decreases exponentially or faster in both the target size and the deviation size. T-TBS is simple and highly scalable when applicable, but only works under the strong restriction that the mean data arrival rate is known and constant. There are scenarios where T-TBS might be a good choice (see Section 3), but many applications have non-constant, unknown mean arrival rates or cannot tolerate sample overflows.

R-TBS: We then provide a novel algorithm, Reservoir-Based Time-Biased Sampling (R-TBS), that is the first to simultaneously enforce (1) at all times, provide a guaranteed upper bound on the sample size, and allow unknown, varying data arrival rates. Guaranteed bounds are desirable because they avoid memory management issues associated with sample overflows, especially when large numbers of samples are being maintained—so that the probability of *some* sample overflowing is high—or when sampling is being performed in a limited memory setting such as at the “edge” of the IoT. Also, bounded samples reduce variability in retraining times and do not impose upper limits on the incoming data flow.

The idea behind R-TBS is to adapt the classic reservoir sampling algorithm, which bounds the sample size but does not allow time biasing. Our approach rests on the notion of a “fractional” sample whose nonnegative size is real-valued in an appropriate sense. We show that, over all sampling algorithms having exponential decay, R-TBS maximizes the expected sample size whenever the data arrival rate is low and also minimizes the sample-size variability.

Distributed implementation: Both T-TBS and R-TBS can be parallelized. Whereas T-TBS is relatively straightforward to implement, an efficient distributed implementation of R-TBS is nontrivial. We exploit various implementation strategies to reduce I/O relative to other approaches, avoid unnecessary concurrency control, and make decentralized decisions about which items to insert into, or delete from, the reservoir.

Organization: The rest of the paper is organized as follows. In Section 2 we formally describe our batch-arrival problem setting and discuss two prior simple sampling schemes: a simple Bernoulli scheme as in [27] and the classical reservoir sampling scheme, modified for batch arrivals. These methods either bound the sample size but do not control the decay rate, or control the decay rate but not the sample size. We next present and analyze the T-TBS and R-TBS algorithms in Section 3 and Section 4. We describe the distributed implementation in Section 5, and Section 6 contains experimental results. We review the related literature in Section 7 and conclude in Section 8.

2 SETTING AND PRIOR SCHEMES

After introducing our problem setting, we discuss two prior sampling schemes that provide context for our current work: simple Bernoulli time-biased sampling (B-TBS) with no sample-size control and the classical reservoir sampling algorithm (with no time biasing), modified for batch arrivals (B-RS).

Setting: Items arrive in *batches* $\mathcal{B}_1, \mathcal{B}_2, \dots$, at time points $t = 1, 2, \dots$, where each batch contains 0 or more items. This

simple integer batch sequence often arises from the discretization of time [24, 28]. Specifically, the continuous time domain is partitioned into intervals of length Δ , and the items are observed only at times $\{k\Delta : k = 0, 1, 2, \dots\}$. All items that arrive in an interval $[k\Delta, (k+1)\Delta)$ are treated as if they arrived at time $k\Delta$, i.e., at the start of the interval, so that all items in batch \mathcal{B}_i have time stamp $i\Delta$, or simply time stamp i if time is measured in units of length Δ . As discussed below, our results can straightforwardly be extended to arbitrary real-valued batch-arrival times.

Our goal is to generate a sequence $\{S_t\}_{t \geq 0}$, where S_t is a sample of the items that have arrived at or prior to time t , i.e., a sample of the items in $U_t = S_0 \cup (\bigcup_{i=1}^t \mathcal{B}_i)$. Here we allow the initial sample S_0 to start out nonempty. These samples should be biased towards recent items so as to enforce (1) for $i \in \mathcal{B}_t$ and $j \in \mathcal{B}_{t'}$ while keeping the sample size as close as possible to (and preferably never exceeding) a specified target n .

Our assumption that batches arrive at integer time points can easily be dropped. In all of our algorithms, inclusion probabilities—and, as discussed later, closely related item “weights”—are updated at a batch arrival time t' with respect to their values at the previous time $t = t' - 1$ via multiplication by $e^{-\lambda}$. To extend our algorithms to handle arbitrary successive batch arrival times t and t' , we simply multiply instead by $e^{-\lambda(t'-t)}$. Thus our results can be applied to arbitrary sequences of real-valued batch arrival times, and hence to an arbitrary sequences of item arrivals (since batches can comprise single items).

Bernoulli Time-Biased Sampling (B-TBS): In the simplest sampling scheme, at each time t , we accept each incoming item $x \in \mathcal{B}_t$ into the sample with probability 1. At each subsequent time $t' > t$, we flip a coin independently for each item currently in the sample: an item is retained in the sample with probability $p = e^{-\lambda}$ and removed with probability $1 - p$. It is straightforward to adapt the algorithm to batch arrivals; see Appendix A of [16], where we show that $\Pr[x \in S_{t'}] = e^{-\lambda(t'-t)}$ for $x \in \mathcal{B}_t$, implying (1). This is essentially the algorithm used, e.g., in [27] to implement time-biased edge sampling in dynamic graphs. The user, however, cannot independently control the expected sample size, which is completely determined by λ and the sizes of the incoming batches. In particular, if the batch sizes systematically grow over time, then sample size will grow without bound. Arguments in [27] show that if $\sup_t |\mathcal{B}_t| < \infty$, then the sample size can be bounded, but only probabilistically. See Remark 1 below for extensions and refinements of these results.

Batched Reservoir Sampling (B-RS): The classic reservoir sampling algorithm can be modified to handle batch arrivals; see Appendix B of [16]. Although B-RS guarantees an upper bound on the sample size, it does not support time biasing. The R-TBS algorithm (Section 4) maintains a bounded reservoir as in B-RS while simultaneously allowing time-biased sampling.

3 TARGETED-SIZE TBS

As a first step towards time-biased sampling with a controlled sample size, we describe the simple T-TBS scheme, which improves upon the simple Bernoulli sampling scheme B-TBS by ensuring the inclusion property in (1) while providing probabilistic guarantees on the sample size. We require that the mean batch size equals a constant b that is both known in advance and “large enough” in that $b \geq n(1 - e^{-\lambda})$, where n is the target sample size and λ is the decay rate as before. The requirement on b ensures that, at the target sample size, items arrive on average at least as fast as they decay.

Algorithm 1: Targeted-size TBS (T-TBS)

```

1  $\lambda$ : decay factor ( $\geq 0$ )
2  $n$ : target sample size
3  $b$ : assumed mean batch size such that  $b \geq n(1 - e^{-\lambda})$ 
4 Initialize:  $S \leftarrow S_0$ ;  $p \leftarrow e^{-\lambda}$ ;  $q \leftarrow n(1 - e^{-\lambda})/b$ 
5 for  $t \leftarrow 1, 2, \dots$  do
6    $m \leftarrow \text{BINOMIAL}(|S|, p)$  //simulate  $|S|$  trials
7    $S \leftarrow \text{SAMPLE}(S, m)$  //retain  $m$  random elements
8    $k \leftarrow \text{BINOMIAL}(|\mathcal{B}_t|, q)$ 
9    $B'_t \leftarrow \text{SAMPLE}(\mathcal{B}_t, k)$  //down-sample new batch
10   $S \leftarrow S \cup B'_t$ 
11  output  $S$ 

```

The pseudocode is given as Algorithm 1. T-TBS is similar to B-TBS in that we downsample by performing a coin flip for each item with retention probability p . Unlike B-TBS, we downsample the incoming batches at rate $q = n(1 - e^{-\lambda})/b$, which ensures that n becomes the “equilibrium” sample size. Specifically, when the sample size equals n , the expected number $n(1 - e^{-\lambda})$ of current items deleted at an update equals the expected number qb of inserted new items, which causes the sample size to drift towards n . Arguing similarly to Appendix A of [16], we have for $t' \geq t \geq 1$ and $x \in \mathcal{B}_t$ that $\Pr[x \in S_{t'}] = qe^{-\lambda(t'-t)}$, so that the key relative appearance property in (1) holds.

For efficiency, the algorithm exploits the fact that for k independent trials, each having success probability r , the total number of successes has a binomial distribution with parameters k and r . Thus, in lines 6 and 8, the algorithm simulates the coin tosses by directly generating the number of successes m or k —which can be done using standard algorithms [17]—and then retaining m or k randomly chosen items. So the function $\text{BINOMIAL}(j, r)$ returns a random sample from the binomial distribution with j independent trials and success probability r per trial, and the function $\text{SAMPLE}(A, m)$ returns a uniform random sample, without replacement, containing $\min(m, |A|)$ elements of the set A ; note that the function call $\text{SAMPLE}(A, 0)$ returns an empty sample for any empty or nonempty A .

Theorem 3.1 below precisely describes the behavior of the sample size; the proof—along with the proofs of most other results in the paper—is given in Appendix C of [16]. Denote by $B_t = |\mathcal{B}_t|$ the (possibly random) size of \mathcal{B}_t for $t \geq 1$ and by $C_t = |S_t|$ the sample size at time t for $t \geq 0$; assume that C_0 is a finite deterministic constant. Define the *upper-support ratio* for a random batch size B as $r = b^*/b \geq 1$, where $b = E[B]$ and b^* is the smallest positive number such that $P[B \leq b^*] = 1$; set $r = \infty$ if B can be arbitrarily large. For $r \in [1, \infty)$, set

$$v_{\epsilon, r}^+ = (1 + \epsilon) \ln((1 + \epsilon)/r) - (1 + \epsilon - r).$$

for $\epsilon > 0$ and

$$v_{\epsilon, r}^- = (1 - \epsilon) \ln((1 - \epsilon)/r) - (1 - \epsilon - r)$$

for $\epsilon \in (0, 1)$. Note that $v_{\epsilon, r}^+ > 0$ and is strictly increasing in ϵ for $\epsilon > r - 1$, and that $v_{\epsilon, r}^-$ increases from $r - 1 - \ln r$ to r as ϵ increases from 0 to 1. Write “i.o.” to denote that an event occurs “infinitely often”, i.e., for infinitely many values of t , and write “w.p.1” for “with probability 1”.

THEOREM 3.1. *Suppose that the batch sizes $\{B_t\}_{t \geq 1}$ are i.i.d with common mean $b \geq n(1 - e^{-\lambda})$, finite variance, and upper support ratio r . Then, for any $p = e^{-\lambda} < 1$,*

- (i) for all $m \geq 0$, we have $\Pr[C_t = m \text{ i.o.}] = 1$;
- (ii) $E[C_t] = n + p^t(C_0 - n)$ for $t > 0$;

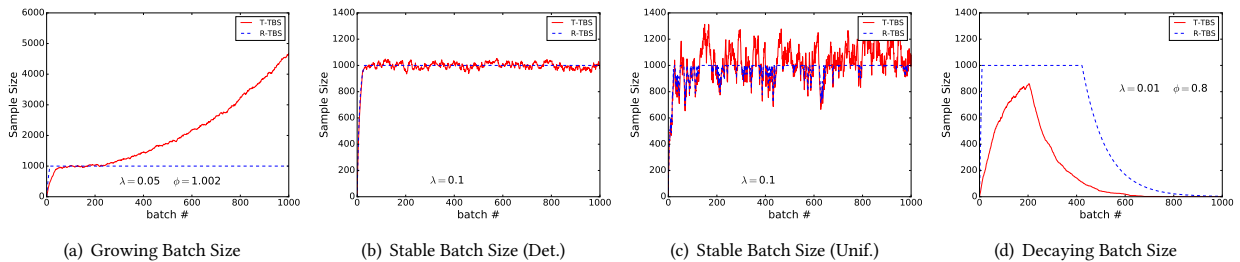


Figure 1: Targeted TBS: Sample Size Behavior, $\lambda =$ decay rate and $\phi =$ batch size multiplier.

- (iii) $\lim_{t \rightarrow \infty} (1/t) \sum_{i=0}^t C_i = n$ w.p.1;
 - (iv) if $C_0 = n$ and $r < \infty$, then
 - (a) $\Pr[C_t \geq (1 + \epsilon)n] \leq e^{-n v_{\epsilon, r}^+ (1 + O(n \epsilon p^t))}$ and
 - (b) $\Pr[C_t \leq (1 - \epsilon)n] \leq e^{-n v_{\epsilon, r}^- (1 + O(n(1 - \epsilon)p^t))}$
- for (a) $\epsilon, t > 0$ and (b) $\epsilon \in (0, 1)$ and $t \geq \ln \epsilon / \ln p$.

In Appendix C of [16], we actually prove a stronger version of the theorem in which the assumption in (iv) that $r < \infty$ is dropped.

Thus, from (ii), $\lim_{t \rightarrow \infty} E[C_t] = n$ so that the expected sample size converges to the target size n as t becomes large; indeed, if $C_0 = n$ then the expected sample size equals n for all $t > 0$. By (iii), an even stronger property holds in that, w.p.1, the average sample size—averaged over the first t batch-arrival times—converges to n as t becomes large. For typical batch-size distributions, the assertions in (iv) imply that, at any given time t , the probability that the sample size deviates from n by more than 100% decreases exponentially with n and—in the case of a positive deviation as in (iv)(a)—super-exponentially in ϵ . However, the assertion in (i) implies that any sample size m , no matter how large, will be exceeded infinitely often w.p.1; indeed, it follows from the proof that the mean times between successive exceedances are not only finite, but are uniformly bounded over time. In summary, the sample size is generally stable and close to n on average, but is subject to infrequent, but unboundedly large spikes in the sample size, so that sample-size control is incomplete.

Indeed, when batch sizes fluctuate in a non-predictable way, as often happens in practice, T-TBS can break down; see Figure 1, in which we plot sample sizes for T-TBS and, for comparison, R-TBS. The problem is that the value of the mean batch size b must be specified in advance, so that the algorithm cannot handle dynamic changes in b without losing control of either the decay rate or the sample size.

In Figure 1(a), for example, the (deterministic) batch size is initially fixed and the algorithm is tuned to a target sample size of 1000, with a decay rate of $\lambda = 0.05$. At $t = 200$, the batch size starts to increase (with $B_{t+1} = \phi B_t$ where $\phi = 1.002$), leading to an overflowing sample, whereas R-TBS maintains a constant sample size.

Even in a stable batch-size regime with constant batch sizes (or, more generally, small variations in batch size), R-TBS can maintain a constant sample size whereas the sample size under T-TBS fluctuates in accordance with Theorem 3.1; see Figure 1(b) for the case of a constant batch size $B_t \equiv 100$ with $\lambda = 0.1$.

Large variations in the batch size lead to large fluctuations in the sample size for T-TBS; in this case the sample size for R-TBS is bounded above by design, but large drops in the batch size can cause drops in the sample size for both algorithms; see Figure 1(c) for the case of $\lambda = 0.1$ and i.i.d. uniformly distributed batch sizes on $[0, 200]$ so that $E[B_t] \equiv 100$. Similarly, as shown in Figure 1(d), systematically decreasing batch sizes will cause the

sample size to shrink for both T-TBS and R-TBS. Here, $\lambda = 0.01$ and, as with Figure 1(a), the batch size is initially fixed and then starts to change at time $t = 200$, with $\phi = 0.8$ in this case. This experiment—and others, not reported here, with varying values of λ and ϕ —indicate that R-TBS is more robust to sample underflows than T-TBS.

Overall, however, T-TBS is of interest because, when the mean batch size is known and constant over time, and when some sample overflows are tolerable, T-TBS is simple to implement and parallelize, and is very fast (see Section 6). For example, if the data comes from periodic polling of a set of robust sensors, the data arrival rate will be known a priori and will be relatively constant, except for the occasional sensor failure, and hence T-TBS might be appropriate. On the other hand, if data is coming from, e.g., a social network, then batch sizes may be hard to predict.

REMARK 1. When $q = 1$, Theorem 3.1 provides a description of sample-size behavior for B-TBS. Under the conditions of the theorem, the expected sample size converges to $n = b/(1 - e^{-\lambda})$, which illustrates that the sample size and decay rate cannot be controlled independently. The actual sample size fluctuates around this value, with large deviations above or below being exponentially or super-exponentially rare. Thus Theorem 3.1 both complements and refines the analysis in [27].

4 RESERVOIR-BASED TBS

Targeted time-biased sampling (T-TBS) controls the decay rate but only partially controls the sample size, whereas batched reservoir sampling (B-RS) bounds the sample size but does not allow time biasing. Our new reservoir-based time-biased sampling algorithm (R-TBS) combines the best features of both, controlling the decay rate while ensuring that the sample never overflows and has optimal sample size and stability properties. Importantly, unlike T-TBS, the R-TBS algorithm can handle any sequence of batch sizes.

4.1 The R-TBS Algorithm

To maintain a bounded sample, R-TBS combines the use of a reservoir with the notion of item *weights*. In R-TBS, the weight of an item initially equals 1 but then decays at rate λ , i.e., the weight of an item $i \in \mathcal{B}_t$ at time $t' \geq t$ is $w_{t'}(i) = e^{-\lambda(t'-t)}$. All items arriving at the same time have the same weight, so that the *total weight* of all items seen up through time t is $W_t = \sum_{j=1}^t B_j e^{-\lambda(t-j)}$, where, as before, $B_j = |\mathcal{B}_j|$ is the size of the j th batch.

R-TBS generates a sequence of latent “fractional samples” $\{L_t\}_{t \geq 0}$ such that (i) the “size” of each L_t equals the *sample weight* C_t , defined as $C_t = \min(n, W_t)$, and (ii) L_t contains $\lfloor C_t \rfloor$ “full” items and at most one “partial” item. For example, a latent sample of size $C_t = 3.6$ contains three “full” items that belong to the actual sample S_t with probability 1 and one partial item that

Algorithm 2: Reservoir-based TBS (R-TBS)

```

1  $\lambda$ : decay factor ( $\geq 0$ )
2  $n$ : maximum sample size
3 Initialize:  $A \leftarrow A_0; W \leftarrow C \leftarrow |A_0|; \pi \leftarrow \emptyset$  //  $|A_0| \leq n$ 
4 for  $t \leftarrow 1, 2, \dots$  do
5   if  $W < n$  then //has been unsaturated
6      $W \leftarrow e^{-\lambda} W$  //decay current items
7     if  $W > 0$  then
8        $(A, \pi, C) \leftarrow \text{DSAMPLE}((A, \pi, C), W)$ 
9        $A \leftarrow A \cup \mathcal{B}_t$  //accept all items in  $\mathcal{B}_t$ 
10       $W \leftarrow W + |\mathcal{B}_t|$  //update total weight
11      if  $W > n$  then //sample is now saturated
12        //adjust for overshoot
13         $(A, \pi, C) \leftarrow \text{DSAMPLE}((A, \pi, W), n)$ 
14      else //has been saturated
15         $W \leftarrow e^{-\lambda} W + |\mathcal{B}_t|$  //new total weight
16        if  $W \geq n$  then //still saturated
17           $m \leftarrow \text{STOCHROUND}(|\mathcal{B}_t|n/W)$ 
18          //replace  $m$   $A$ -items with  $m$   $\mathcal{B}_t$ -items
19           $A \leftarrow A \setminus \text{SAMPLE}(A, m) \cup \text{SAMPLE}(\mathcal{B}_t, m)$ 
20        else //now unsaturated
21          //adjust for undershoot
22           $(A, \pi, C) \leftarrow \text{DSAMPLE}((A, \pi, n), W - |\mathcal{B}_t|)$ 
23           $A \leftarrow A \cup \mathcal{B}_t$  //all batch items are full
24     $S \leftarrow \text{GETSAMPLE}(A, \pi, C)$ 
25  output  $S$ 

```

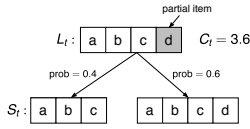


Figure 2: Latent sample L_t (sample weight $C_t = 3.6$) and possible realized samples.

belongs to S_t with probability 0.6. Thus S_t is obtained by including each full item and then including the partial item according to its associated probability, so that C_t represents the expected size of S_t . E.g., in our example, the sample S_t will contain either three or four items with respective probabilities 0.4 and 0.6, so that the expected sample size is 3.6; see Figure 2. Note that if $C_t = k$ for some $k \in \{0, 1, \dots, n\}$, then with probability 1 the sample contains precisely k items, and C_t is the actual size of S_t , rather than just the expected size. Since each C_t by definition never exceeds n , no sample S_t ever contains more than n items.

More precisely, given a set U of items, a *latent sample* of U with sample weight C is a triple $L = (A, \pi, C)$, where $A \subseteq U$ is a set of $\lfloor C \rfloor$ *full* items and $\pi \subseteq U$ is a (possibly empty) set containing at most one *partial* item. At each time t , we randomly generate S_t from $L_t = (A_t, \pi_t, C_t)$ by sampling such that

$$S_t = \begin{cases} A_t \cup \pi & \text{with probability } \text{frac}(C_t); \\ A_t & \text{with probability } 1 - \text{frac}(C_t), \end{cases} \quad (2)$$

where $\text{frac}(x) = x - \lfloor x \rfloor$. That is, each full item is included with probability 1 and the partial item is included with probability $\text{frac}(C_t)$. Thus

$$\begin{aligned} \mathbb{E}[|S_t|] &= \lceil C_t \rceil \text{frac}(C_t) + \lfloor C_t \rfloor (1 - \text{frac}(C_t)) \\ &= (\lceil C_t \rceil - \lfloor C_t \rfloor) \text{frac}(C_t) + \lfloor C_t \rfloor \\ &= \text{frac}(C_t) + \lfloor C_t \rfloor = C_t \end{aligned} \quad (3)$$

as previously asserted. By allowing at most one partial item, we minimize the latent sample’s footprint: $|A_t \cup \pi_t| \leq \lfloor C_t \rfloor + 1$.

The key goal of R-TBS is to maintain the invariant

$$\Pr[i \in S_t] = (C_t/W_t)w_t(i) \quad (4)$$

for each $t \geq 0$ and each item $i \in U_t$, where, as before, U_t denotes the set of all items that arrive up through time t , so that the appearance probability for an item i at time t is proportional to its weight $w_t(i)$. This immediately implies the desired relative-inclusion property (1). Since $w_t(i) = 1$ for an arriving item $i \in \mathcal{B}_t$, the equality in (4) implies that the initial acceptance probability for this item is

$$\Pr[i \in S_t] = C_t/W_t. \quad (5)$$

The pseudocode for R-TBS is given as Algorithm 2. Suppose the sample is *unsaturated* at time $t - 1$ in that $W_{t-1} < n$ and hence $C_{t-1} = W_{t-1}$ (line 5). The decay process first reduces the total weight (and hence the sample weight) to $W'_{t-1} = C'_{t-1} = e^{-\lambda}W_{t-1}$ (line 6). R-TBS then *downsamples* L_{t-1} (line 8) to reflect this decay and maintain a minimal sample footprint; the downsampling method, described in Section 4.2, is designed to maintain the invariant in (4). If the weight of the arriving batch does not cause the sample to overflow, i.e., $C'_{t-1} + |\mathcal{B}_t| < n$, then $C_t = C'_{t-1} + |\mathcal{B}_t| = W'_{t-1} + |\mathcal{B}_t| = W_t$. The relation in (5) then implies that all newly arrived items are accepted into the sample with probability 1 (line 9); see Figure 3(a) for an example of this scenario. The situation is more complicated if the weight of the arriving batch would cause the sample to overflow. It turns out that the simplest way to deal with this scenario is to initially accept all incoming items as in line 9, and then run an additional round of downsampling to reduce the sample weight to n (line 12), so that the sample is now saturated; see Figure 3(b). Note that these two steps can be executed without ever causing the sample footprint to exceed n .

Now suppose that the sample is *saturated* at time $t - 1$, so that $W_{t-1} \geq n$ and hence $C_{t-1} = |S_{t-1}| = n$. The new total weight is $W_t = W'_{t-1} + |\mathcal{B}_t|$ as before (line 14). If $W_t \geq n$, then the weight of the arriving batch exceeds the weight loss due to decay, and the sample remains saturated. Then (5) implies that each item in \mathcal{B}_t is accepted into the sample with probability $p = n/W_t$. Letting $I_j = 1$ if item $j \in \mathcal{B}$ is accepted and $I_j = 0$ otherwise, we see that the expected number of accepted items is

$$m = \mathbb{E}\left[\sum_{j \in \mathcal{B}_t} I_j\right] = \sum_{j \in \mathcal{B}_t} \mathbb{E}[I_j] = \sum_{j \in \mathcal{B}_t} \Pr[I_j = 1] = B_t n/W_t.$$

There are a number of possible ways to carry out this acceptance operation, e.g., via independent coin flips. To minimize the variability of the sample size (and hence the likelihood of severely small samples), R-TBS uses *stochastic rounding* in line 16 and accepts a random number of items M such that $M = \lfloor m \rfloor$ with probability $\lceil m \rceil - m$ and $M = \lceil m \rceil$ with probability $m - \lfloor m \rfloor$, so that $\mathbb{E}[M] = m$ by an argument essentially the same as in (3). To maintain the bound on the sample size, the M accepted items replace M randomly selected “victims” in the current sample (line 17). If $W_t < n$, then the sample weight decays to W'_{t-1} and the weight of the arriving batch is not enough to fill the sample back up. Moreover, (5) implies that all arriving items are accepted with probability 1. Thus we downsample to the decayed weight of $W'_{t-1} = W_t - |\mathcal{B}_t|$ in line 19 and then insert the arriving items in line 20.

4.2 Downsampling

Before describing Algorithm 3, the downsampling algorithm, we intuitively motivate a key property that any such procedure must have. For any item $i \in L$, the relation in (4) implies that we must have $\Pr[i \in S] = (C/W)w_i$ and $\Pr[i \in S'] = (C'/W')w'_i$, where W and w_i represent the total and item weight before decay and

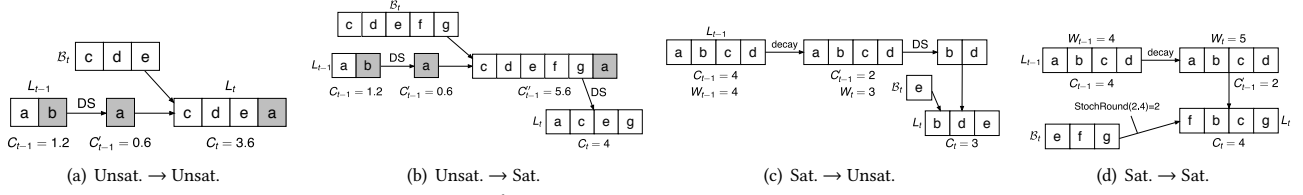


Figure 3: R-TBS scenarios for $n = 4$ and $e^{-\lambda} = 0.5$. For simplicity, we take $W_{t-1} = C_{t-1}$. “DS” denotes downsampling.

Algorithm 3: Downsampling

```

1  $L = (A, \pi, C)$ : input latent sample
2  $C'$ : input target weight with  $0 < C' < C$ 
3  $L' = (A', \pi', C')$ : output latent sample
4  $U \leftarrow \text{UNIFORM}()$ 
5 if  $\lfloor C' \rfloor = 0$  then //no full items retained
6   if  $U > \text{frac}(C)/C$  then
7      $(A', \pi') \leftarrow \text{SWAP1}(A, \pi)$ 
8      $A' \leftarrow \emptyset$ 
9 else if  $0 < \lfloor C' \rfloor = \lfloor C \rfloor$  then //no items deleted
10  if  $U > (1 - (C'/C) \text{frac}(C)) / (1 - \text{frac}(C'))$  then
11     $(A', \pi') \leftarrow \text{SWAP1}(A, \pi)$ 
12 else //items deleted:  $0 < \lfloor C' \rfloor < \lfloor C \rfloor$ 
13  if  $U \leq (C'/C) \text{frac}(C)$  then
14     $A' \leftarrow \text{SAMPLE}(A, \lfloor C' \rfloor)$ 
15     $(A', \pi') \leftarrow \text{SWAP1}(A', \pi)$ 
16  else
17     $A' \leftarrow \text{SAMPLE}(A, \lfloor C' \rfloor + 1)$ 
18     $(A', \pi') \leftarrow \text{MOVE1}(A', \pi)$ 
19 if  $C' = \lfloor C' \rfloor$  then //no fractional item
20    $\pi' \leftarrow \emptyset$ 

```

downsampling, and W' and w'_i represent the weights afterwards. Since decay affects all items equally, we have $w/W = w'/W'$, and it follows that

$$\Pr[i \in S'] = (C'/C) \Pr[i \in S]. \quad (6)$$

That is, the inclusion probabilities for all items must be scaled down by the same fraction, namely C'/C . Theorem 4.1 (later in this section) asserts that Algorithm 3 satisfies this property.

In the pseudocode for Algorithm 3, the function $\text{UNIFORM}()$ generates a random number uniformly distributed on $[0, 1]$. The subroutine $\text{SWAP1}(A, \pi)$ moves a randomly selected item from A to π and moves the current item in π (if any) to A . Similarly, $\text{MOVE1}(A, \pi)$ moves a randomly selected item from A to π , replacing the current item in π (if any). More precisely, $\text{SWAP1}(A, \pi)$ executes the operations $I \leftarrow \text{SAMPLE}(A, 1)$, $A \leftarrow (A \setminus I) \cup \pi$, and $\pi \leftarrow I$, and $\text{MOVE1}(A, \pi)$ executes the operations $I \leftarrow \text{SAMPLE}(A, 1)$, $A \leftarrow A \setminus I$, and $\pi \leftarrow I$.

To gain some intuition for why the algorithm works, consider a simple special case, where the goal is to form a fractional sample $L' = (A', \pi', C')$ from a fractional sample $L = (A, \pi, C)$ of integral size $C > C'$; that is, L comprises exactly C full items. Assume that C' is non-integral, so that L' contains a partial item. In this case, we simply select an item at random (from A) to be the partial item in L' and then select $\lfloor C' \rfloor$ of the remaining $C - 1$ items at random to be the full items in L' ; see Figure 4(a). By symmetry, each item $i \in L$ is equally likely to be included in S' , so that the inclusion probabilities for the items in L are all scaled down by the same fraction, as required for (6). For example, taking $t = 0$ in Figure 4(a), item a appears in S_t with probability 1 since it is a full item. In S'_t , where the weights have been reduced by 50%, item a (either as a full or partial item, depending on the random outcome) appears with probability $2 \cdot (1/6) + 2 \cdot (1/6) \cdot 0.5 = 0.5$, as expected. This scenario corresponds to lines 17 and 18 in the

algorithm, where we carry out the above selections by randomly sampling $\lfloor C' \rfloor + 1$ items from A to form A' and then choosing a random item in A' as the partial item by moving it to π .

In the case where L contains a partial item i^* that appears in S with probability $\text{frac}(C)$, it follows from (6) that i^* should appear in S' with probability $p = (C'/C)P[i^* \in S] = (C'/C) \text{frac}(C)$. Thus, with probability p , lines 13–15 retain i^* and convert it to a full item so that it appears in S' . Otherwise, in lines 17 and 18, i^* is removed from the sample when it is overwritten by a random item from A' ; see Figure 4(b). Again, a new partial item is chosen from A in a random manner to uniformly scale down the inclusion probabilities. For instance, in Figure 4(b), item d appears in S_t with probability 0.2 (because it is a partial item) and in S'_t appears with probability $3 \cdot (0.1/3) = 0.1$. Similarly, item a appears in S_t with probability 1 and in S'_t with probability $(1.8)/6 + 0.6 \cdot (1.8/6) + 0.6 \cdot (0.1/3) = 0.5$.

The if-statement in line 5 corresponds to the corner case in which L' does not contain a full item. The partial item $i^* \in L$ either becomes full or is swapped into A' and then immediately ejected; see Figure 4(c).

The if-statement in line 9 corresponds to the case in which no items are deleted from the latent sample, e.g., when $C = 4.7$ and $C' = 4.2$. In this case, i^* either becomes full by being swapped into A' or remains as the partial item for L' . Denoting by ρ the probability of *not* swapping, we have $P[i^* \in S'] = \rho \cdot \text{frac}(C') + (1 - \rho) \cdot 1$. On the other hand, (6) implies that $P[i^* \in S'] = (C'/C) \text{frac}(C)$. Equating these expressions shows that ρ must equal the expression on the right side of the inequality on line 10; see Figure 4(d).

Formally, we have the following result.

THEOREM 4.1. For $0 < C' < C$, let $L' = (A', \pi', C')$ be the latent sample produced from a latent sample $L = (A, \pi, C)$ via Algorithm 3, and let S' and S be samples produced from L' and L via (2). Then $\Pr[i \in S'] = (C'/C) \Pr[i \in S]$ for all $i \in L$.

4.3 Properties of R-TBS

Theorem 4.2 below asserts that R-TBS satisfies (4) and hence (1), thereby maintaining the correct inclusion probabilities; see Appendix C of [16] for the proof. Theorems 4.3 and 4.4 assert that, among all sampling algorithms with exponential time biasing, R-TBS both maximizes the expected sample size in unsaturated scenarios and minimizes sample-size variability. Thus R-TBS tends to yield more accurate results (from more training data) and greater stability in both result quality and retraining costs.

THEOREM 4.2. The relation $\Pr[i \in S_t] = (C_t/W_t)w_t(i)$ holds for all $t \geq 1$ and $i \in U_t$.

THEOREM 4.3. Let H be any sampling algorithm that satisfies (1) and denote by S_t and S_t^H the samples produced at time t by R-TBS and H . If the total weight at some time $t \geq 1$ satisfies $W_t < n$, then $E[|S_t^H|] \leq E[|S_t|]$.

PROOF. Since H satisfies (1), it follows that, for each time $j \leq t$ and $i \in \mathcal{B}_j$, the inclusion probability $\Pr[i \in S_t^H]$ must be of the

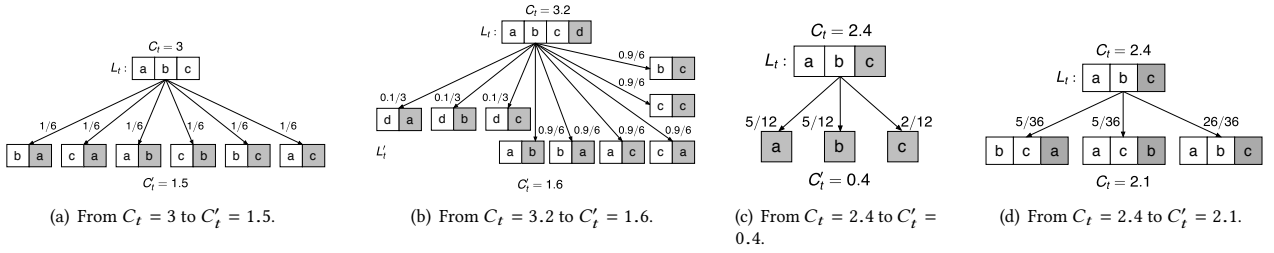


Figure 4: Downsampling examples ($t = 0$).

form $r_t e^{-\lambda(t-j)}$ for some function r_t independent of j . Taking $j = t$, we see that $r_t \leq 1$. For R-TBS in an unsaturated state, (4) implies that $r_t = C_t/W_t = 1$, so that $\Pr[i \in S_t^H] \leq \Pr[i \in S_t]$, and the desired result follows directly. \square

THEOREM 4.4. *Let H be any sampling algorithm that satisfies (1) and has maximal expected sample size C_t and denote by S_t and S_t^H the samples produced at time t by R-TBS and H . Then $\text{Var}[|S_t^H|] \geq \text{Var}[|S_t|]$ for any time $t \geq 1$.*

PROOF. Considering all possible distributions over the sample size having a mean value equal to C_t , it is straightforward to show that variance is minimized by concentrating all of the probability mass onto $\lfloor C_t \rfloor$ and $\lceil C_t \rceil$. There is precisely one such distribution, namely the stochastic-rounding distribution, and this is precisely the sample-size distribution attained by R-TBS. \square

5 DISTRIBUTED TBS ALGORITHMS

In this section, we describe how to implement distributed versions of T-TBS and R-TBS to handle large volumes of data.

5.1 Overview of Distributed Algorithms

The distributed T-TBS and R-TBS algorithms, denoted as D-T-TBS and D-R-TBS respectively, need to distribute large data sets across the cluster and parallelize the computation on them.

Overview of D-T-TBS: The implementation of the D-T-TBS algorithm is very similar to the simple distributed Bernoulli time-biased sampling algorithm in [27]. It is embarrassingly parallel, requiring no coordination. At each time point t , each worker in the cluster subsamples its partition of the sample with probability p , subsamples its partition of \mathcal{B}_t with probability q , and then takes a union of the resulting data sets.

Overview of D-R-TBS: This algorithm, unlike D-T-TBS, maintains a bounded sample, and hence cannot be embarrassingly parallel. D-R-TBS first needs to aggregate local batch sizes to compute the incoming batch size $|\mathcal{B}_t|$ to maintain the total weight W . Then, based on $|\mathcal{B}_t|$ and the previous total weight W , D-R-TBS determines whether the reservoir was previously saturated and whether it will be saturated after processing \mathcal{B}_t . For each possible situation, D-R-TBS chooses the items in the reservoir to delete through downsampling and the items in \mathcal{B}_t to insert into the reservoir. This process requires the master to coordinate among the workers. In Section 5.3, we introduce two alternative approaches to determine the deleted and inserted items. Finally, the algorithm applies the deletes and inserts to form the new reservoir, and computes the new total weight W .

Both D-T-TBS and D-R-TBS periodically checkpoint the sample as well as other system state variables to ensure fault tolerance. The implementation details for D-T-TBS are mostly subsumed by those for D-R-TBS, so we focus on the latter.

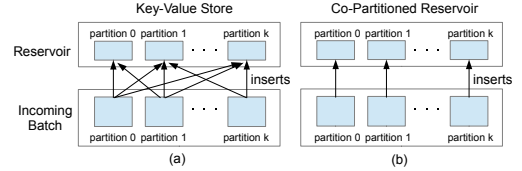


Figure 5: Design choices for implementing the reservoir

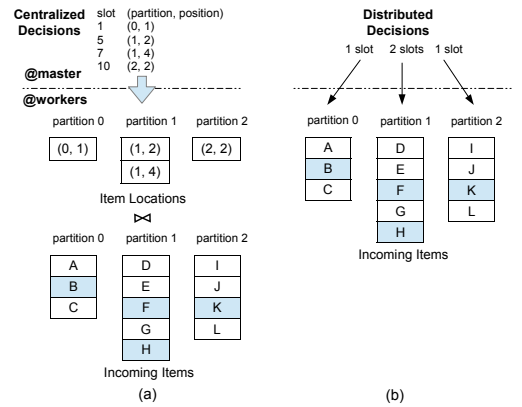


Figure 6: Retrieving insert items

5.2 Distributed Data Structures

There are two important data structures in the D-R-TBS algorithm: the incoming batch and the reservoir. Conceptually, we view an incoming batch \mathcal{B}_t as an array of slots numbered from 1 through $|\mathcal{B}_t|$, and the reservoir as an array of slots numbered from 1 through $\lfloor C \rfloor$ containing full items plus a special slot for the partial item. For both data structures, data items need to be distributed into partitions due to the large data volumes. Therefore, the slot number of an item maps to a specific partition ID and a position inside the partition.

The incoming batch usually comes from a distributed streaming system, such as Spark Streaming; the actual data structure is specific to the streaming system (e.g. an incoming batch is stored as an RDD in Spark Streaming). As a result, the partitioning strategy of the incoming batch is opaque to the D-R-TBS algorithm. Unlike the incoming batch, which is read-only and discarded at the end of each time period, the reservoir data structure must be continually updated. An effective strategy for storing and operating on the reservoir is thus crucial for good performance. We now explore alternative approaches to implementing the reservoir.

Distributed in-memory key-value store: One quite natural approach implements the reservoir using an off-the-shelf distributed in-memory key-value store, such as Redis [25] or Memcached [23]. In this scheme, each item in the reservoir is stored as a key-value pair, with the slot number as the key and the item as the value. Inserts and deletes to the reservoir naturally translate into put and delete operations to the key-value store.

There are two major limitations to this approach. Firstly, the hash-based or range-based data-partitioning scheme used by a distributed key-value store yields reservoir partitions that do not correlate with the partitions of incoming batch. As illustrated in Figure 5(a), when items from a given partition of an incoming batch are inserted into the reservoir, the inserts touch many (if not all) partitions of the reservoir, incurring heavy network I/O. Secondly, key-value stores incur needless concurrency-control overhead. For each batch, D-R-TBS already carefully coordinates the deletes and inserts so that no two delete or insert operations access the same slots in the reservoir and there is no danger of write-write or read-write conflicts.

Co-partitioned reservoir: In the alternative approach, we implement a distributed in-memory data structure for the reservoir so as to ensure that the reservoir partitions coincide with the partitions from incoming batches, as shown in Figure 5(b). This can be achieved in spite of the unknown partitioning scheme of the streaming system. Specifically, the reservoir is initially empty, and all items in the reservoir are from the incoming batches. Therefore, if an item from a given partition of an incoming batch is always inserted into the corresponding “local” reservoir partition and deletes are also handled locally, then the co-partitioning and co-location of the reservoir and incoming batch partitions is automatic. For our experiments, we implemented the co-partitioned reservoir in Spark using the in-place updating technique for RDDs in [27]; see Appendix E of [16].

Note that, at any point in time, a given slot number in the reservoir maps to a specific partition ID and a position inside the partition. Thus the slot number for a given full item may change over time due to reservoir insertions and deletions. This does not cause any statistical issues, because the functioning of the set-based R-TBS algorithm is oblivious to specific slot numbers.

5.3 Choosing Items to Delete and Insert

In order to bound the reservoir size, D-R-TBS requires careful coordination when choosing the set of items to delete from, and insert into, the reservoir. At the same time, D-R-TBS must ensure the statistical correctness of random number generation and random permutation operations in the distributed environment. We consider two possible approaches.

Centralized decisions: In the most straightforward approach, the master makes centralized decisions about which items to delete and insert. For deletes, the driver generates slot numbers of the items in the reservoir to be deleted, which are then mapped to the actual data locations in a manner that depends on the representation of the reservoir (key-value store or co-partitioned reservoir). For inserts, the driver generates the slot numbers of the incoming items \mathcal{B}_t at time t that need to be inserted into the reservoir. Suppose that \mathcal{B}_t comprises $k \geq 1$ partitions. Each generated slot number $i \in \{1, 2, \dots, |\mathcal{B}_t|\}$ is mapped to a partition p_i of the \mathcal{B}_t (where $0 \leq p_i \leq k - 1$) and a position r_i inside partition p_i . Denote by Q the set of “item locations”, i.e., the set of (p_i, r_i) pairs. In order to perform the inserts, we need to first retrieve the actual items based on the item locations. This can be achieved with a join-like operation between Q and \mathcal{B}_t , with the (p_i, r_i) pair matching the actual location of an item inside \mathcal{B}_t . To optimize this operation, we make Q a distributed data structure and use a customized partitioner to ensure that all pairs (p_i, r_i) with $p_i = j$ are co-located with partition j of \mathcal{B}_t for $j = 0, 1, \dots, k - 1$. Then a co-partitioned and co-located join can be carried out between Q and \mathcal{B}_t , as illustrated in Figure 6(a)

for $k = 3$. The resulting set of retrieved insert items, denoted as S , is also co-partitioned with \mathcal{B}_t as a by-product. After that, the actual deletes and inserts are then carried out depending on how reservoir is stored, as discussed below.

When the reservoir is implemented as a key-value store, the deletes can be directly applied based on the slot numbers. For inserts, the master takes each generated slot number of an item in \mathcal{B}_t and chooses a companion destination slot number in the reservoir into which the \mathcal{B}_t item will be inserted. This destination reservoir slot might currently be empty due to an earlier deletion, or might contain an item that will now be replaced by the newly inserted batch item. After the actual items to insert are retrieved as described previously, the destination slot numbers are used to put the items into the right locations in the key-value store.

When the co-partitioned reservoir is used, the delete slot numbers in the reservoir are mapped to (p_i, r_i) pairs of partitions of the reservoir and positions inside the partitions. As with inserts, we again use a customized partitioner for the set of pairs \mathcal{R} such that deletes are co-located with the corresponding reservoir partitions. Then a join-like operation on \mathcal{R} and the reservoir performs the actual delete operations on the reservoir. For inserts, we simply use another join-like operation on the set of retrieved insert items S and the reservoir to add the corresponding insert items to the co-located partition of the reservoir. In this approach, we don’t need the master to generate destination reservoir slot numbers for these insert items, because we view the reservoir as a set when using co-partitioned reservoir data structure.

Distributed decisions: The above approach requires generating a large number of slot numbers inside the master, so we now explore an alternative approach that offloads the slot number generation to the workers while still ensuring the statistical correctness of the computation. This approach has the master choose only the number of deletes and inserts per worker according to appropriate multivariate hypergeometric distributions. For deletes, each worker chooses random victims from its local partition of the reservoir based on the number of deletes given by the master. For inserts, the worker randomly and uniformly selects items from its local partition of the incoming batch \mathcal{B}_t given the number of inserts. Figure 6(b) depicts how the insert items are retrieved under this decentralized approach. We use the technique in [15] for parallel pseudo-random number generation.

Note that this distributed decision making approach works only when the co-partitioned reservoir data structure is used. This is because the key-value store representation of the reservoir requires a target reservoir slot number for each insert item from the incoming batch, and the target slot numbers have to be generated in such a way as to ensure that, after the deletes and inserts, all of the slot numbers are still unique and contiguous in the new reservoir. This requires a lot of coordination among the workers, which inhibits truly distributed decision making.

6 EXPERIMENTS

In this section, we study the empirical performance of D-R-TBS and D-T-TBS, and demonstrate the potential benefit of using them for model retraining in online model management. We implemented D-R-TBS and D-T-TBS on Spark (refer to Appendix E of [16] for implementation details).

Experimental Setup: All performance experiments were conducted on a cluster of 13 IBM System x iDataPlex dx340 servers. Each has two quad-core Intel Xeon E5540 2.8GHz processors and 32GB of RAM. Servers are interconnected using a 1Gbit Ethernet and each server runs Ubuntu Linux, Java 1.7 and Spark 1.6.

One server is dedicated to run the Spark coordinator and, each of the remaining 12 servers runs Spark workers. There is one worker per processor on each machine, and each worker is given all 4 cores to use, along with 8 GB of dedicated memory. All other Spark parameters are set to their default values. We used Memcached 1.4.33 as the key-value store in our experiments.

For all experiments, data was streamed in from HDFS using Spark Streaming’s microbatches. We report run time per round as the average over 100 rounds, discarding the first round from this average because of Spark startup costs. Unless otherwise stated, each batch contains 10 million items, the target reservoir size is 20 million elements, and the decay parameter is $\lambda = 0.07$.

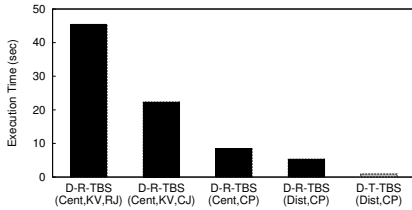


Figure 7: Per-batch distributed runtime comparison

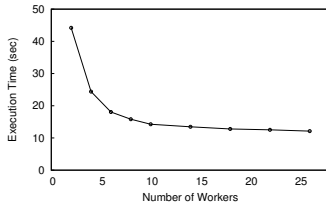


Figure 8: Scale out of D-R-TBS

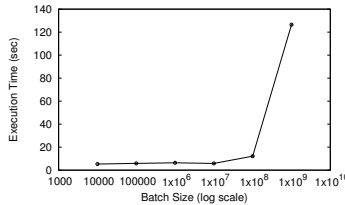


Figure 9: Scale up of D-R-TBS

6.1 Runtime Performance

Comparison of TBS Implementations: Figure 7 shows the average runtime per batch for five different implementations of distributed TBS algorithms. The first four (colored black) are D-R-TBS implementations with different design choices: whether to use centralized or distributed decisions (abbreviated as “Cent” and “Dist”, respectively) for choosing items to delete and insert, and whether to use key-value store for storing reservoir or co-partitioned reservoir (abbreviated as “KV” and “CP”, respectively). The first two implementations both use the key-value store representation for reservoir together with the centralized decision strategy for determining inserts and deletes. They only differ in how the insert items are actually retrieved when subsampling the incoming batch. The first uses the standard repartition join (abbreviated as “RJ”), whereas the second uses the customized partitioner and co-located join (abbreviated as “CJ”) as described in Section 5.3 and depicted in Figure 6(a). This optimization effectively cuts the network cost in half, but the KV representation of reservoir still requires the insert items to be written across the network to their corresponding reservoir location. The third implementation employs the co-partitioned reservoir instead, resulting in a significant speedup of over 2.6x. The fourth implementation further employs the distributed decision for choosing items to delete and insert. This yields a further 1.6x speedup. We use this D-R-TBS implementation in the remaining experiments.

The fifth implementation (colored grey) in Figure 7 is D-T-TBS using co-partitioned reservoir and the distributed strategy for choosing delete and insert items. Since, D-T-TBS is embarrassingly parallelizable, it’s much faster than the best D-R-TBS implementation. But, as we discussed in Section 3, T-TBS only works under a very strong restriction on the data arrival rate,

and can suffer from occasional memory overflows; see Figure 1. In contrast, D-R-TBS is much more robust and works in realistic scenarios where it is hard to predict the data arrival rate.

Scalability of D-R-TBS: Figure 8 shows how D-R-TBS scales with the number of workers. We increased the batch size to 100 million items for this experiment. Initially, D-R-TBS scales out very nicely with the increasing number of workers. However, beyond 10 workers, the marginal benefit from additional workers is small, because the coordination and communication overheads, as well as the inherent Spark overhead, become prominent. For the same reasons, in the scale-up experiment in Figure 9, the runtime stays roughly constant until the batch size reaches 10 million items and increases sharply at 100 million items. This is because processing the streaming input and maintaining the sample start to dominate the coordination and communication overhead. With 10 workers, R-TBS can handle a data flow comprising 100 million items arriving approximately every 14 seconds.

6.2 Application: Classification using kNN

We now demonstrate the potential benefits of the R-TBS sampling scheme for periodically retraining representative ML models in the presence of evolving data. For each model and data set, we compare the quality of models retrained on the samples generated by R-TBS, a simple sliding window (SW), and uniform reservoir sampling (Unif). Due to limited space, we do not give quality results for T-TBS; we found that whenever it applies—i.e. when the mean batch size is known and constant—the quality is very similar to R-TBS, since they both use time-biased sampling.

Our first model is a kNN classifier, where a class is predicted for each item in an incoming batch by taking a majority vote of the classes of the k nearest neighbors in the current sample, based on Euclidean distance; the sample is then updated using the batch. To generate training data, we first generate 100 class centroids uniformly in a $[0, 80] \times [0, 80]$ rectangle. Each data item is then generated from a Gaussian mixture model and falls into one of the 100 classes. Over time, the data generation process operates in one of two “modes”. In the “normal” mode, the frequency of items from any of the first 50 classes is five times higher than that of items in any of the second 50 classes. In the “abnormal” mode, the frequencies are five times lower. Thus the frequent and infrequent classes switch roles at a mode change. We generate each data point by randomly choosing a ground-truth class c_i with centroid (x_i, y_i) according to relative frequencies that depend upon the current mode, and then generating the data point’s (x, y) coordinates independently as samples from $N(x_i, 1)$ and $N(y_i, 1)$. Here $N(\mu, \sigma)$ denotes the normal distribution with mean μ and standard deviation σ .

In this experiment, the batch sizes are deterministic with $b = 100$ items, and $k = 7$ neighbors for the kNN classifier. The reservoir size for both R-TBS and Unif is 1000, and SW contains the last 1000 items; thus all methods use the same amount of data for retraining. (We choose this value because it achieves near maximal classification accuracies for all techniques. In general, we choose sampling and ML parameters to achieve good learning performance while ensuring fair comparisons.) In each run, the sample is warmed up by processing 100 normal-mode batches before the classification task begins. Our experiments focus on two types of temporal patterns in the data, as described below.

Single change: Here we model the occurrence of a singular event. The data is generated in normal mode up to $t = 10$ (time is measured here in units after warm-up), then switches to abnormal

mode, and finally at $t = 20$ switches back to normal (Figure 10(a)). As can be seen, the misclassification rate (percentage of incorrect classifications) with R-TBS, SW and Unif all increase from around 18% to roughly 50% when the distribution becomes abnormal. Both R-TBS and SW adapt to the change, recovering to around 16% misclassification rate after $t = 16$, with SW adapting slightly better. In comparison, Unif does not adapt at all. But, when the distribution snaps back to normal, the error rate of SW rises sharply to 40% before gradually recovering, whereas R-TBS error rate stays low around 15% throughout. These results prove that R-TBS is indeed more robust: although slightly more sluggish than SW in adapting to changes, R-TBS avoids wild fluctuations in classification error as with SW.

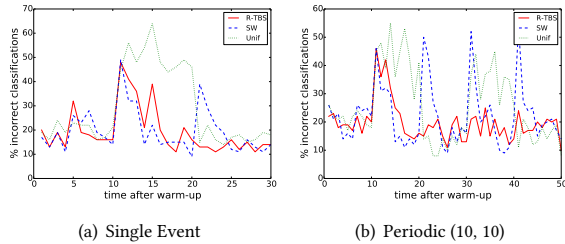


Figure 10: Misclassification rate (percent) for kNN

Periodic change: For this temporal pattern, the changes from normal to abnormal mode are periodic, with δ normal batches alternating with η abnormal batches, denoted as $\text{Periodic}(\delta, \eta)$, or $P(\delta, \eta)$ for short. Figure 10(b) shows the misclassification rate for $\text{Periodic}(10, 10)$. Experiments on other periodic patterns (in Appendix F of [16]) demonstrate similar results. The robust behavior of R-TBS described above manifests itself even more clearly in the periodic setting. Note, for example, how R-TBS reacts significantly better to the renewed appearances of the abnormal mode. Observe that the first 30 batches of $\text{Periodic}(10, 10)$ display the same behavior as in the single event experiment in Figure 10(a). We therefore focus primarily on the $\text{Periodic}(10, 10)$ temporal pattern for the remaining experiments.

Robustness and Effect of Decay Parameter: In the context of online model management, we need a sampling scheme that delivers high overall prediction accuracy and, perhaps even more importantly, robust prediction performance over time. Large fluctuations in the accuracy can pose significant risks in applications, e.g., in critical IoT applications in the medical domain such as monitoring glucose levels for predicting hyperglycemia events. To assess the robustness of the performance results across different sampling schemes, we use a standard risk measure called *expected shortfall (ES)* [22, p. 70]. ES measures downside risk, focusing on worst-case scenarios. Specifically, the $z\%$ ES is the average value of the worst $z\%$ of cases.

For each of 30 runs and for each sampling scheme, we compute the 10% ES of the misclassification rate (expressed as a percentage) starting from $t = 20$, since all three sampling schemes perform poorly (as would be expected) during the first mode change, which finishes at $t = 20$. Table 1 lists both the *accuracy*, measured in terms of the average misclassification rate, and the *robustness*, measured as the average 10% ES, of the kNN classifier over 30 runs across different temporal patterns. To demonstrate the effect of the decay parameter λ on model performance, we also include numbers for different λ values in Table 1.

In terms of accuracy, Unif is always the worst by a large margin. R-TBS and SW have similar accuracies, with R-TBS having a

slight edge in most cases. On the other hand, for robustness, SW is almost always the worst, with ES ranging from 1.4x to 2.7x the maximum ES (over different λ values) of R-TBS. Mostly, Unif is also significantly worse than R-TBS, with ES ratios ranging from 1.4x to 1.7x. The only exception is the single-event pattern: since the data remains in normal mode after the abnormal period, time biasing becomes unimportant and Unif performs well. In general, R-TBS provides both better accuracy and robustness in almost all cases. The relative performance of the sampling schemes in terms of accuracy and robustness tend to be consistent across temporal patterns. Table 1 also shows that different λ values affect the accuracy and robustness, however, R-TBS provides superior results over a fairly wide range of λ values.

Varying batch size: We now examine model quality when the batch sizes are no longer constant. Overall, the results look similar to those for constant batch size. For example, Figure 11(a) shows results for a $\text{Uniform}(0,200)$ batch-size distribution, and Figure 11(b) shows results for a deterministic batch size that grows at a rate of 2% after warm-up. In both experiments, $\lambda = 0.07$ and the data pattern is $\text{Periodic}(10, 10)$. These figures demonstrate the robust performance of R-TBS in the presence of varying data arrival rates. Similarly, the average accuracy and robustness over 30 runs resembles the results in Table 1. For example, pick $\lambda = 0.07$ and a $\text{Periodic}(10, 10)$ pattern. Then, the misclassification rate under uniform/growing batch sizes is 1.16x/1.14x that of R-TBS for SW, and 1.47x/1.40x for Unif. In addition, the ES is 1.82x/1.98x that of R-TBS for SW, and 1.76x/1.78x for Unif.

Table 1: Accuracy and robustness of kNN performance

λ	Single Event		P(10,10)		P(20,10)		P(30,10)	
	Miss%	ES	Miss%	ES	Miss%	ES	Miss%	ES
0.05	19.8	17.7	18.2	24.2	17.9	28.2	15.5	31.6
0.07	19.1	18.7	17.4	23.2	17.2	28.1	14.9	31.0
0.10	18.0	20.0	16.6	24.1	16.6	29.9	15.1	31.0
SW	19.2	53.3	19.0	49.8	18.8	47.3	16.5	44.5
Unif	25.6	19.3	25.4	42.3	25.0	43.2	21.0	47.6

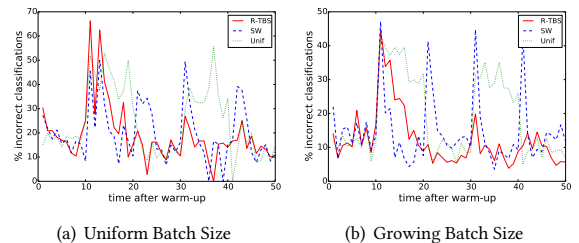


Figure 11: Varying batch sizes for kNN classifier

6.3 Application: Linear Regression

We now assess the effectiveness of R-TBS for retraining regression models. The experimental setup is similar to kNN, with data generated in “normal” and “abnormal” modes. In both modes, data items are generated from the standard linear regression model $y = b_1x_1 + b_2x_2 + \epsilon$, with the noise term ϵ distributed according to a $N(0, 1)$ distribution. In normal mode, $(b_1, b_2) = (4.2, -0.4)$ and in abnormal mode, $(b_1, b_2) = (-3.6, 3.8)$. In both modes, x_1 and x_2 are generated according to $\text{Uniform}(0, 1)$ distribution. As before, the experiment starts with a warm-up of 100 “normal” mode batches and each batch contains 100 items.

Saturated samples: Figure 12(a) shows the performance of R-TBS, SW, and Unif for the $\text{Periodic}(10, 10)$ pattern with a maximum sample size of 1000 for each technique. We note that, for this sample size and temporal pattern, the R-TBS sample is always saturated. (This is also true for all of the prior experiments.) The results echo that of the previous section, with R-TBS exhibiting

slightly better prediction accuracy on average, and significantly better robustness, than the other methods. The mean square errors (MSEs) across all data points for R-TBS, Unif, and SW are 3.51, 4.43, 4.02 respectively, and their 10% ES of the MSEs are 6.04, 10.05, 10.94 respectively.

Unsaturated Samples: We now investigate the case of unsaturated samples for R-TBS. We increase the target sample size to $n = 1600$. With a constant batch size of 100, and a decay rate $\lambda = 0.07$, the reservoir of R-TBS is never full, stabilizing at 1479 items, whereas Unif and SW both have a full sample of 1600 items.

For the Periodic(10, 10) pattern, shown in Figure 12(b), SW has a window size large enough to keep some data from older time periods (up to 16 batches ago), making SW’s robustness comparable to R-TBS (ES of 5.86 for SW and 5.97 for R-TBS). However, this amalgamation of old data also hurts its overall accuracy, with MSE rising to 4.17, as opposed to 3.50 for R-TBS. In comparison, the shape of R-TBS remains almost unchanged from Figure 12(a), and Unif behaves as poorly as before. When the pattern changes to Periodic(16, 16) as shown in Figure 12(c), SW doesn’t contain enough old data, making its prediction performance suffer from huge fluctuations again, and the superiority of R-TBS is more prominent. In both cases, R-TBS provides the best overall performance, despite having a smaller sample size. This backs up our earlier claim that more data is not always better. A smaller but more balanced sample with good ratios of old and new data can provide better prediction performance than a large but unbalanced sample.

6.4 Application: Naive Bayes

In our final experiment, we evaluate the performance of R-TBS for retraining Naive Bayes models with the Usenet2 dataset (mlkd.csd.auth.gr/concept_drift.html), which was used in [18] to study classifiers coping with recurring contexts in data streams. This dataset contains a stream of 1500 messages on different topics from the 20 News Groups Collections [21]. They are sequentially presented to a simulated user who marks whether a message is interesting or not. The user’s interest changes after every 300 messages. More details of the dataset can be found in [18].

Following [18], we use Naive Bayes with a bag of words model, and set the optimal parameters for SW with maximum sample size of 300 and batch size of 50. Since this dataset is rather small and contexts change frequently, we use the optimal value of 0.3 for λ . We find through experiments that R-TBS displays higher prediction accuracy for all λ in the range of [0.1, 0.5], so precise tuning of λ is not critical. In addition, there is not enough data to warm up the models on different sampling schemes, so we report the model performance on all the 30 batches. Similarly, we report 20% ES for this dataset, due to the limited number of batches.

The results are shown in Figure 13. The misprediction rate for R-TBS, SW, and Unif are 26.5%, 30.0%, and 29.5%; and the 20% ES values are 43.3%, 52.7%, and 42.7%. Importantly, for this dataset the changes in the underlying data patterns are less pronounced than in the previous two experiments. Despite this, SW fluctuates wildly, yielding inferior accuracy and robustness. In contrast, Unif barely reacts to the context changes. As a result, Unif is very slightly better than R-TBS with respect to robustness, but at the price of lower overall accuracy. Thus, R-TBS is generally more accurate under mild fluctuations in data patterns, and its superior robustness properties manifest themselves as the changes become more pronounced.

7 RELATED WORK

Time-decay and sampling: Work on sampling with unequal probabilities goes back to at least Lahiri’s 1951 paper [20]. A growing interest in streaming scenarios with weighted and decaying items began in the mid-2000’s, with most of that work focused on computing specific aggregates from such streams, such as heavy-hitters, subset sums, and quantiles; see, e.g., [2, 7, 8]. The first papers on time-biased reservoir sampling with exponential decay are due to Aggarwal [1] and Efraimidis and Spirakis [12]; batch arrivals are not considered in these works. As discussed in Section 1, the sampling schemes in [1] are tied to item sequence numbers rather than the wall clock times on which we focus; the latter are more natural when dealing with time-varying data arrival rates.

Cormode et al. [9] propose a time biased reservoir sampling algorithm based on the A-Res weighted sampling scheme proposed in [12]. Rather than enforcing (1), the algorithm enforces the (different) A-Res biasing scheme. In more detail, if s_i denotes the element at slot i in the reservoir, then the algorithm in [12] implements a scheme where an item x is chosen to be at slot $i + 1$ in the reservoir with probability $w_x / (\sum_{j=1}^x w_j - \sum_{j=1}^i w_{s_j})$. From the form of this equation, it becomes clear that resulting sampling algorithm violates (1). Indeed, Efraimidis [11] gives some numerical examples illustrating this point (in his comparison of the A-Res and A-Chao algorithms). Again, we would argue that the constraint on appearance probabilities in (1) is easier to understand in the setting of model management than the foregoing constraint on initial acceptance probabilities.

The closest solution to ours adapts the weighted sampling algorithm of Chao [5] to batches and time decay; we call the resulting algorithm B-Chao and describe it in Appendix D of [16]. Unfortunately, as discussed, the relation in (1) is violated both during the initial fill-up phase and whenever the data arrival rate becomes slow relative to the decay rate, so that the sample contains “overweight” items. Including overweight items causes over-representation of older items, thus potentially degrading predictive accuracy. The root of the issue is that the sample size is nondecreasing over time. The R-TBS algorithm is the first algorithm to correctly (and optimally) deal with “underflows” by allowing the sample to shrink—thus handling data streams whose flow rates vary unrestrictedly over continuous time. The current paper also explicitly handles batch arrivals and explores parallel implementation issues. The VarOpt sampling algorithm of Cohen et al. [6]—which was developed to solve the specific problem of estimating “subset sums”—can also be modified to our setting. The resulting algorithm is more efficient than Chao, but as stated in [6], it has the same statistical properties, and hence does not satisfy (1).

Model management: A key goal of our work is to support model management; see [13] for a survey on methods for detecting changing data—also called “concept drift” in the setting of online learning—and for adapting models to deal with drift. As mentioned previously, one possibility is to re-engineer the learning algorithm. This has been done, for example, with support-vector machines (SVMs) by developing incremental versions of the basic SVM algorithm [4] and by adjusting the training data in an SVM-specific manner, such as by adjusting example weights as in Klinkenberg [19]. Klinkenberg also considers using curated data selection to learn over concept drift, finding that weighted data selection also improves the performance of learners. Our approach of model retraining using time-biased samples follows this

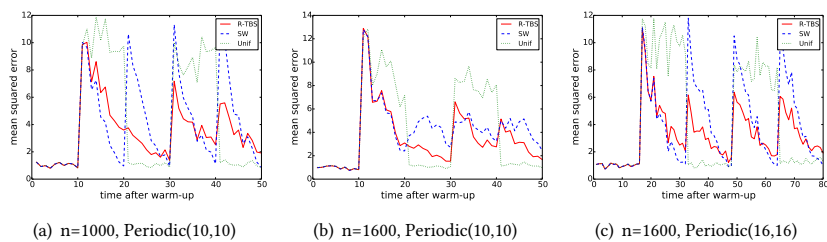


Figure 12: Mean square error for linear regression

latter approach, and is appealing in that it is simple and applies to a large class of machine-learning models. The recently proposed Velox system for model management [10] ties together online learning and statistical techniques for detecting concept drift. After detecting drift through poor model performance, Velox kicks off batch learning algorithms to retrain the model. Our approach to model management is complementary to the work in [10] and could potentially be used in a system like Velox to help deployed models recover from poor performance more quickly. The developers of the recent MacroBase system [3] have incorporated a time-biased sampling approach to model retraining, for identifying and explaining outliers in fast data streams. MacroBase essentially uses Chao’s algorithm, and so could potentially benefit from the R-TBS algorithm to enforce the inclusion criterion (1) in the presence of highly variable data arrival rates.

8 CONCLUSION

Our experiments with classification and regression algorithms, together with the prior work on graph analytics in [27], indicate the potential usefulness of periodic retraining over time-biased samples to help ML algorithms deal with evolving data streams without requiring algorithmic re-engineering. To this end we have developed and analyzed several time-biased sampling algorithms that are of independent interest. In particular, the R-TBS algorithm allows simultaneous control of both the item-inclusion probabilities and the sample size, even when the data arrival rate is unknown and can vary arbitrarily. R-TBS also maximizes the expected sample size and minimizes sample-size variability over all possible bounded-size algorithms with exponential decay. Using techniques from [9], we intend to generalize these properties of R-TBS to hold under arbitrary forms of temporal decay.

We have also provided techniques for distributed implementation of R-TBS and T-TBS, and have shown that use of time-biased sampling together with periodic model retraining can improve model robustness in the face of abnormal events and periodic behavior in the data. In settings where (i) the mean data arrival rate is known and (roughly) constant, as with a fixed set of sensors, and (ii) occasional sample overflows can be easily dealt with by allocating extra memory, we recommend use of T-TBS to precisely control item-inclusion probabilities. In many applications, however, we expect that either (i) or (ii) will be violated, in which case we recommend the use of R-TBS. Our experiments showed that R-TBS is superior to sliding windows over a range of λ values, and hence does not require highly precise parameter tuning; this may be because time-biased sampling avoids the all-or-nothing item inclusion mechanism inherent in sliding windows.

REFERENCES

[1] Charu C. Aggarwal. 2006. On biased reservoir sampling in the presence of stream evolution. In *VLDB*. VLDB Endowment, 607–618.
[2] Noga Alon, Nick Duffield, Carsten Lund, and Mikkel Thorup. 2005. Estimating arbitrary subset sums with few probes. In *PODS*. ACM, 317–325.

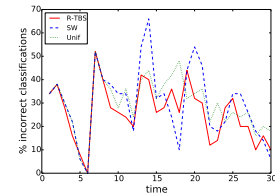


Figure 13: Misclassification rate (percent) for Naive Bayes

[3] Peter Bailis, Edward Gan, Samuel Madden, Deepak Narayanan, Kexin Rong, and Sahaana Suri. 2017. MacroBase: Prioritizing Attention in Fast Data. In *SIGMOD*. 541–556.
[4] Gert Cauwenberghs and Tomaso Poggio. 2000. Incremental and Decremental Support Vector Machine Learning. In *NIPS*. 388–394.
[5] M. T. Chao. 1982. A general purpose unequal probability sampling plan. *Biometrika* (1982), 653–656.
[6] Edith Cohen, Nick G. Duffield, Haim Kaplan, Carsten Lund, and Mikkel Thorup. 2011. Efficient Stream Sampling for Variance-Optimal Estimation of Subset Sums. *SIAM J. Comput.* 40, 5 (2011), 1402–1431.
[7] Edith Cohen and Martin J Strauss. 2006. Maintaining time-decaying stream aggregates. *J. Algo.* 59, 1 (2006), 19–36.
[8] Graham Cormode, Flip Korn, and Srikanta Tirathapura. 2008. Exponentially decayed aggregates on data streams. In *ICDE*. IEEE, 1379–1381.
[9] Graham Cormode, Vladislav Shkapenyuk, Divesh Srivastava, and Bojian Xu. 2009. Forward decay: A practical time decay model for streaming systems. In *ICDE*. IEEE, 138–149.
[10] Daniel Crankshaw, Peter Bailis, Joseph E. Gonzalez, Haoyuan Li, Zhao Zhang, Michael J. Franklin, Ali Ghodsi, and Michael I. Jordan. 2015. The Missing Piece in Complex Analytics: Low Latency, Scalable Model Management and Serving with Velox. In *CIDR*.
[11] Pavlos S. Efraimidis. 2015. Weighted Random Sampling over Data Streams. In *Algorithms, Probability, Networks, and Games*, Christos D. Zaroliagis, Grammati E. Pantziou, and Spyros C. Kontogiannis (Eds.). Springer, 183–195.
[12] Pavlos S Efraimidis and Paul G Spirakis. 2006. Weighted random sampling with a reservoir. *Inf. Process. Lett.* 97, 5 (2006), 181–185.
[13] João Gama, Indre Zliobaite, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. 2014. A survey on concept drift adaptation. *ACM Comput. Surv.* 46, 4 (2014), 44.
[14] Rainer Gemulla and Wolfgang Lehner. 2008. Sampling time-based sliding windows in bounded space. In *SIGMOD*. 379–392.
[15] Hiroshi Haramoto, Makoto Matsumoto, Takuji Nishimura, François Panneton, and Pierre L’Ecuyer. 2008. Efficient Jump Ahead for 2-Linear Random Number Generators. *INFORMS Journal on Computing* 20(3) (2008), 385–390.
[16] Brian Hentschel, Peter J. Haas, and Yuanyuan Tian. 2018. Temporally-Biased Sampling for Online Model Management. *CoRR* abs/1801.09709 (2018). <https://arxiv.org/abs/1801.09709>
[17] Voratas Kachitvichyanukul and Bruce W. Schmeiser. 1988. Binomial Random Variate Generation. *Commun. ACM* 31, 2 (1988), 216–222.
[18] Ioannis Katakis, Grigoris Tsoumakas, and I Vlahavas. 2008. An Ensemble of Classifiers for coping with Recurring Contexts in Data Streams. (01 2008), 763–764 pages.
[19] Ralf Klinkenberg. 2004. Learning drifting concepts: Example selection vs. example weighting. *Intell. Data Anal.* 8, 3 (2004), 281–300.
[20] D. B. Lahiri. 1951. A method of sample selection providing unbiased ratio estimates. *Bull. Intl. Statist. Inst.* 33 (1951), 133–140.
[21] M. Lichman. 2013. UCI Machine Learning Repository. (2013). <http://archive.ics.uci.edu/ml>
[22] Alexander J. McNeil, Rüdiger Frey, and Paul Embrechts. 2015. *Quantitative Risk Management: Concepts, Techniques and Tools* (second ed.).
[23] Memcached. 2017. (2017). Retrieved 2017-07-13 from <https://memcached.org>
[24] Zhengping Qian, Yong He, Chunzhi Su, Zhuojie Wu, Hongyu Zhu, Taizhi Zhang, Lidong Zhou, Yuan Yu, and Zheng Zhang. 2013. TimeStream: Reliable stream computation in the cloud. In *EuroSys*.
[25] Redis. 2017. (2017). Retrieved 2017-07-13 from <https://redis.io>
[26] Andrew Whitmore, Anurag Agarwal, and Li Da Xu. 2015. The Internet of Things – A survey of topics and trends. *Information Systems Frontiers* 17, 2 (2015), 261–274.
[27] Wenlei Xie, Yuanyuan Tian, Yannis Sismanis, Andrew Balmin, and Peter J. Haas. 2015. Dynamic interaction graphs with probabilistic edge decay. In *ICDE*. 1143–1154.
[28] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *SOSP*.

Detecting Database File Tampering through Page Carving

James Wagner, Alexander Rasin, Karen Heart,
Tanu Malik, Jacob Furst
DePaul University
Chicago, Illinois
[jwagne32, arasin, kheart, tmalik1, jfurst]@depaul.edu

Jonathan Grier
Grier Forensics
Pikesville, Maryland
jdgrier@grierforensics.com

ABSTRACT

Database Management Systems (DBMSes) secure data against regular users through defensive mechanisms such as access control, and against privileged users with detection mechanisms such as audit logging. Interestingly, these security mechanisms are built into the DBMS and are thus only useful for monitoring or stopping operations that are executed through the DBMS API. Any access that involves directly modifying database files (at file system level) would, by definition, bypass any and all security layers built into the DBMS itself.

In this paper, we propose and evaluate an approach that detects direct modifications to database files that have already bypassed the DBMS and its internal security mechanisms. Our approach applies forensic analysis to first validate database indexes and then compares index state with data in the DBMS tables. We show that indexes are much more difficult to modify and can be further fortified with hashing. Our approach supports most relational DBMSes by leveraging index structures that are already built into the system to detect database storage tampering that would currently remain undetectable.

1 INTRODUCTION

DBMSes use a combination of defense and detection mechanisms to secure access to data. Defense mechanisms, such as access control, determine the data granularity and system access granted to different database users; defense mechanisms, such as audit logging, monitor all database activity. Regardless of the defense mechanisms, security breaches are still a legitimate concern – sometimes due to unintentional granting of extra access control and sometimes due to outright hacking, such as SQL injection. Security breaches are typically detected through analysis of audit logs. However, audit log analysis is unreliable to detect a breach that originated from privileged users.

Privileged users, by definition, already have the ability to control and modify access permissions. Therefore, audit logs fundamentally cannot be trusted to detect suspicious activity. Additionally, privileged users commonly have access to database files. Consider a system administrator who maliciously, acting as the root, edits a DBMS data file in a Hex editor or through a programming language, such as Python. The DBMS, unaware of external file write activity taking place outside its own programmatic access, cannot log it, and thus the tampering attack remains undetected.

Current DBMSes do not provide tools against insider threats – in general, a built-in security mechanism is vulnerable to insider attacks. While a DBMS will not be able to detect direct

storage changes, file-level modifications potentially create inconsistencies within the auxiliary data structures maintained by a DBMS. Forensics tools that examine file contents can be used to detect such inconsistencies, and determine if insider threats have taken place. Recently we proposed the first database forensic tool, DBCarver, that can be used to detect deleted data from database pages [31]. However, database forensic tools such as DBCarver merely extract forensic artifacts but do not search for inconsistencies within the data structures maintained by a DBMS.

In this paper, we propose a system, DBStorage Auditor, that detects database file tampering by identifying inconsistencies in storage through a direct inspection of internal database structures. DBStorage Auditor utilizes existing database forensic techniques and expands them to extract additional necessary storage artifacts. These artifacts are then used to detect inconsistencies within indexes and between indexes and tables. The underlying premise of our approach is that all relational databases follow patterns in storage over which the privileged user has little or no control. We inspect these storage patterns to detect unusual activity. We motivate DBStorage Auditor through an example:

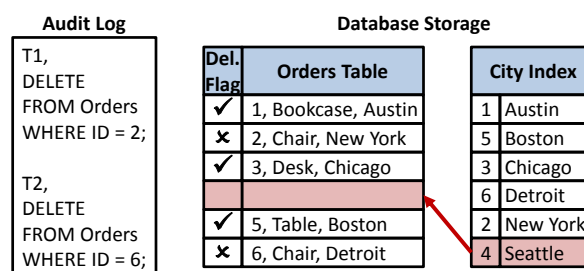


Figure 1: Example attack through DBMS files.

EXAMPLE 1. Malice is the system administrator for a shipping company, FriendlyShipping. Malice is bribed by a competing company to interfere with the orders going to Seattle. Malice **does not** have access to the DBMS, but she **does** have access to the server where the database files reside.

Malice writes a Python script that will open and directly modify the database file containing the Orders table. The script then opens the database file, finds all records containing the string ‘Seattle’, and explicitly overwrites entire records with the NULL ASCII character (decimal value 0).

Figure 1 illustrates the result of Malice’s script actions. Since the record was erased without the DBMS (API has never seen that command) all DBMS security was bypassed, and the operation was never recorded in the log file. When FriendlyShipping investigates the missing Seattle orders, the audit log can only explain deleted orders for (2, Chair, New York) and (6, Chair, Detroit). The audit logs contain no trace of the Seattle order being deleted because it was not deleted but rather wiped out externally.

To simplify in the above example, we have omitted some details of database file tampering, which we expand on later in Section 5. Barring those details in Example 1, the value in the City index

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

still exists in index storage even though the entire record is erased. Therefore, an inconsistency can be identified by mapping back the index value to the empty gap in table storage. The empty gap in table storage exists because a database only marks a record when it is deleted, and only overwrites the record with data from a newly inserted record. However, making the mapping from the index value to the associated record must be based on the behavioral rules of database storage, such as page and record layout. We use database forensic tools to understand database layout, and using that layout, perform the necessary mapping.

It is not impossible for a scrupulous system administrator to (i) tamper with the index and create a cascade of inconsistencies throughout the index structure, or (ii) for an attacker who has privileges to modify database files to acquire privileges to suspend or kill logging mechanisms at the operating system level if necessary, or (iii) for a knowledgeable adversary to easily avoid corrupting storage and keep checksum values consistent. However, in spite of increased level of threat, we repeatedly show that accurate knowledge about data layout can be used to gather evidence and prove if any malicious activity has taken place.

Previously we developed an approach to detect malicious activity when DBMS logging is disabled [28]. In this approach we analyzed unlogged activity (executed through a proper DBMS API) but strictly assumed that database files were not exposed to tampering. In this paper, we address the tampering vulnerability where the database files are physically altered. Developing an auditing system for DBMSes is part of our larger goal to open up the database system and its storage to users, for performance and forensics investigation.

The rest of the paper is organized as follows: Section 2 covers related work. Section 3 discusses concepts of database storage used throughout the paper. Section 4 defines the adversary we seek to defend against. Section 5 details how to perform database file tampering. Section 6 provides an overview of DBStorageAuditor. Section 7 describes how we utilize database forensics. Section 8 addresses index tampering. Section 9 proposes a method to organize carved index output making our system scalable. Section 10 discusses how to detect file tampering using inconsistencies between carved index data and table data. Section 11 provides a thorough evaluation of our system.

2 RELATED WORK

This paper focuses on the detection of database file tampering. Therefore, we discuss work related to protecting DBMSes against privileged users as well as work that detects regular (non-DBMS) file tampering. We outline why existing file tampering and anti-forensic methods are inapplicable to database files.

2.1 Database Auditing and Security

Database audit log files are of great interest for database security because they can be used to determine whether data was compromised and what records were accessed. Methods to verify log integrity have been proposed to detect log file tampering [18, 25]. Pavlou et al. expanded upon this work to determine the time of log tampering [17]. Sinha et al. used hash chains to verify log integrity in an offline environment without requiring communication with a central server [24]. Crosby et al. proposed a data structure, history tree, to reduce the log size produced by hash chains in an offline environment [2]. Rather than detecting log tampering, Schneider and Kelsey developed an approach to make log files impossible to parse and alter [23]. An event log can be generated using triggers, and the idea of a `SELECT` trigger

was explored for the purpose of logging [3]. ManageEngine’s EventLog Analyzer provides audit log reports and alerts for Oracle and SQL Server based on actions, such as user activity, record modification, schema alterations, and read-only queries [13]. We previously described a method to detect inconsistencies between storage and log files, allowing tampering detection when logging was disabled (i.e., when an operation was excluded from the log) [28]. All of this work assumes that database storage can not be altered directly – an action which bypasses logging mechanisms.

Network-based monitoring methods have received attention in audit log research because they provide independence and generality by residing outside of the DBMS. IBM Security Guardium Express Activity Monitor for Databases [9] monitors incoming packets for suspicious activity. Liu et al. [12] monitored DBAs and other privileged users by identifying and logging network packets containing SQL statements. The benefit of monitoring activity over the network and, therefore, beyond the reach of DBA’s, is the level of independence achieved by these tools. On the other hand, relying on network activity ignores local DBMS connections and requires intimate understanding of SQL commands (i.e., an obfuscated command can fool the system).

2.2 Database Forensics

Stahlberg demonstrated the retention of deleted data and proposed techniques to erase data for a MySQL DBMS [26]. While this work was only ever implemented for MySQL, it validates our threat model by imposing custom DBMS file modifications.

Database page carving [31] is a method for reconstructing the contents of a relational database without relying on the file system or DBMS. Page carving is inspired by traditional file carving [6, 21], which reconstructs data (active and deleted) from disk images or RAM snapshots without the need for a live system. The work in [29] presented a comparative study of the page structure for multiple DBMSes. Subsequent work in [30] described how long forensic evidence resides within a database even after being deleted or reorganized. While a multitude of built-in and third party recovery tools (e.g., [15, 19, 20]) aim to extract database storage, none of these tools are helpful for forensic analysis because they can only recover “active” data. Forensic tools, such as Sleuth Kit [1] and EnCASE Forensic [4], are commonly used by digital investigators to reconstruct file system data, but they are not capable of parsing database files. A database forensic tool (just like a forensic file system tool) should also reconstruct unallocated pieces of data, including deleted rows, auxiliary structures (indexes) or buffer cache space.

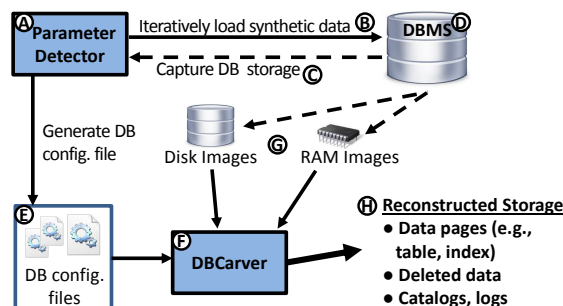


Figure 2: DBCarver architecture.

Our storage analysis relies on DBCarver tool described in [31], which was revised to process additional artifacts for this paper. Figure 2 provides an overview of DBCarver, which consists of two main components: the parameter collector(A) and the carver(F).

The parameter detector loads synthetic data into a DBMS(B), captures storage(C), deconstructs pages from storage, and describes the page layout with a set of parameters which are stored in a configuration file(E) – a text file that captures page-level layout information for that particular DBMS. These configuration files are used by the carver(F) to reconstruct DBMS content from disk images, RAM snapshots, or any other input file(G). The carver returns storage artifacts(H), such as user records, metadata describing user data, deleted data, and system catalogs.

2.3 File Tampering and Anti-Forensics

One-way hash functions have been used to detect file tampering at the file system level [7, 11]. However, we expect database files to be regularly modified by legitimate operations. Distinguishing a malicious tampering operation and a legitimate SQL operation would be nearly impossible at the file system level without knowledge of metadata in DBMS storage. Authenticating cached data on untrusted publishers has been explored by Martel [14] and Tamassia [27]. Their threat model defends against an untrusted publisher that provides cached results working with a trusted DBMS and, while our work addresses an untrusted DBMS.

Anti-forensics is defined as a method that seeks to interfere with a forensic process [8]; file tampering threat model we address in this paper exhibits anti-forensics behavioral properties. Two traditional anti-forensics techniques are data wiping and data hiding [5, 10]: 1) data wiping explicitly overwrites data to delete it rather than mark it as deleted, 2) data hiding seeks to hide the message itself. We are not aware of any existing literature that addresses anti-forensics within DBMSes [22]; we consider adding or erasing data through file tampering (that bypasses DBMS itself) to be the equivalent of anti-forensics for DBMSes.

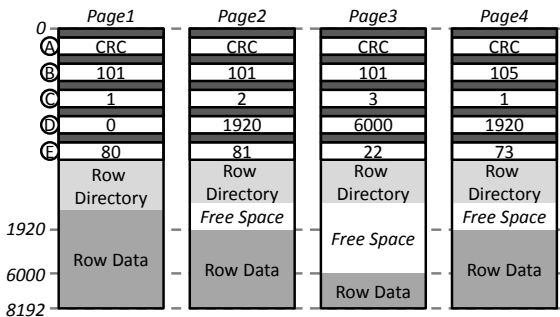


Figure 3: Example page headers.

3 BACKGROUND

The security threats we consider in this paper affect the lowest level of database storage (details of which are hidden from the users by design). In this section, we briefly generalize storage of the RDBMS row-store pages and define terminology used throughout this paper. The concepts formulated in this section apply to (but are not limited to) IBM DB2, SQL Server, Oracle, PostgreSQL, MySQL, Apache Derby, MariaDB, and Firebird.

3.1 Page Layout

When DBMS data is accessed or modified through an API, the DBMS implements data changes within pages and maintains a variety of additional metadata. While each DBMS employs its own storage engine, there are many conceptual commonalities between DBMSes in how data is stored and maintained. Every DBMS uses fixed-size pages with three main structures: header, row directory, and row data.

A DBMS page header stores metadata describing user records stored in the page. The metadata of interest (to this paper) are the checksum, object identifier, page identifier, free space pointer, and record count. Figure 3 demonstrates an example of how this metadata could be positioned in an 8K page. The checksum(A) detects data corruption within a page; whenever a page is modified, the checksum is updated. The object identifier(B) represents the database object to which the page belongs (the object name is stored in a separate system table). In Figure 3, Pages 1-3 have the object identifier 101, and Page 4 has the object identifier 105. The page identifier(C) is unique to each page for either an object, a file, or across all files. In Figure 3, the page identifier is unique for each object because the value 1 occurs for both objects 101 and 105. The free space pointer(D) references unallocated space within the page where a new record can be added. If the page is full, the free space pointer is NULL (decimal value 0). In Figure 3, Page1 is full since it has a NULL free space pointer, while Pages 2, 3, and 4 point to unallocated space. The record count(E) refers to the number of *active* records in a page. If a record is deleted, the record count will be decremented by one, and if a record is added to a page, it will be incremented by one. In Figure 3, Page1 has 80 active records, and Page2 has 81 active records.

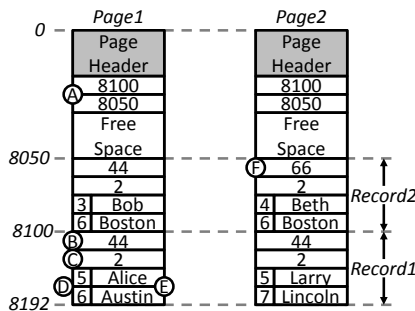


Figure 4: Example row directory and row data layouts.

The row directory stores pointers to each page record (row) – when a record is added to a page, a pointer is added to the row directory. Figure 4 shows one example of how the row directory (A) could be positioned; the row directory in this example has two pointers referencing records within the row data.

The row data stores the user data along with additional metadata. Figure 4 shows an example of how the row data may be structured (with some minor DBMS-specific variations). Each record stores the user data values (E), a row delimiter that separates the records (B), the number of columns for the record (C), and the size of each string (D).

Deleted Data. When a record is deleted, a DBMS either overwrites the row directory pointer for that record or marks the record itself in the row data – it is important to note that the record entry is not erased. Figure 4 shows an example of when the row metadata is marked for a deleted record (F). Deleted records become unallocated space, and DBMS settings and operations dictate when records are (eventually) overwritten by new data.

Index Pages. Index value-pointer pairs are stored in pages, which are similar to table pages including a header, row directory, and row data. The only significant difference between table and index pages is the layout of records – index pages store value-pointer pairs in the row data. Furthermore, in practice, index values are *not* marked as unallocated space when a corresponding table record is deleted. Stale index values persist in storage, typically until the B-Tree is explicitly rebuilt by the user and long after the table record was overwritten.

4 THREAT MODEL

In this section, we define the attack vectors, different possible adversary types, and the privileges we expect them to wield. We consider two types of privileged users: database administrator (DBA) and system administrator (SA). A DBA can issue privileged SQL commands against the DBMS including disabling logs or granting privileges to users. However, a DBA would not have administrative access to the server OS. The SA has administrative access to the server OS including the ability to suspend processes and read/write access to all files, but no access to privileged SQL commands in the DBMS. The SA can still have a regular DB user account without affecting our assumptions.

Since a DBA can bypass DBMS defense mechanisms, detection mechanisms are best suited to identify anomalous behavior. An audit log containing a history of SQL commands is accepted as one of the best detection mechanisms for a DBMS. In Section 2, we discussed prior work designed to prevent audit log tampering and detect malicious behavior in the event that logging was disabled. In this paper, we focus on a detection mechanism for a user often ignored in DBMS security, the SA.

The SA can bypass all DBMS security defense and detection mechanisms by reading and editing a database file with a tool other than the DBMS. For example, a SA could use Python to open a file and change the value 'Hank' to 'Walt'. In Section 5 we discuss additional steps that must be considered to successfully perform such an operation, but it can ultimately be achieved. Since this operation occurs outside of the DBMS, it bypasses all DBMS access control, and it will not be included any of the DBMS log files. Furthermore, one can assume that the SA would have the ability to suspend any logging mechanism in the server OS. Although changes to a file will also be recorded in the file system journal, the SA has the ability to turn off journaling to the file system by using `tune2fs` on Unix or the `FSCTL_DELETE_USN_JOURNAL` control code on NTFS (Windows). However, the file system must be shutdown first in order to prevent possible corruption. Therefore, the SA may have to effect a shutdown of the DBMS before making changes to the database files. The shutting down and restarting of the database instance and the system will generate events that are logged; however, as mentioned earlier, the SA can turn off system logging easily. Moreover, the SA could revise the DBMS log in order to hide evidence of the shutdown and restart. Hence, it would be somewhat involved but not difficult for a SA to cover his/her tracks when tampering with a DBMS file.

5 FILE TAMPERING

The threats to data we consider in this paper occur at the OS level outside of DBMS control. In this section, we formulate the threat and introduce concepts and categories of tampering.

A DBMS allows users and administrators to access and modify data through an API. Access control guarantees that users will be limited to data they are privileged to access. In this section, we discuss how an adversary can perform file tampering. To limit the scope of this paper, we assume that file tampering involves user data and not metadata (changing metadata can easily damage the DBMS but that will not alter any of its records). We define user data as records created by the user or copies of record values that may reside in auxiliary structures (e.g., indexes). File tampering actions that we discuss in this section ultimately produce one of two results in storage: 1) **Extraneous data** is a record or a value that has been added through file tampering or 2) **Erased**

data is a record that has been explicitly overwritten (rather than marked deleted by a command as described in Section 3).

Three things must be considered when performing database file tampering: 1) page checksum, 2) write lock on files, and 3) dirty pages. In Section 3, we discussed the functionality and placing of the page checksum. Figure 5 shows three different page alterations, in all of which the checksum is (also) updated. Some DBMS processes hold write locks on the database files. Therefore, tampering would require that the attacker release or otherwise bypass OS file write locks. DBMSes do not immediately write pages back to disk after they are modified in the buffer cache. That is significant because a maliciously altered page on disk can be overwritten when a dirty page is flushed to disk – or, alternatively, a dirty page could be altered directly in RAM instead (bypassing file locks that way).

Write-Locks. The file locking system API, through the `fcntl` system call in Unix, is set up so that a process can prevent writes to (as well as reads from) a file that it has locked successfully. An attacker can potentially cause the process holding the lock, in this case the DBMS, to release the lock. Otherwise, a sophisticated attacker with root privileges can release the lock without involvement of the process by using kernel code. Once the lock is released, the attacker would lock the file, tamper with its content, and then release the lock. The DBMS would not receive any signal or other indication of the tampering and could continue to use the file as if it were locked after the attacker releases the lock. While the attacker holds the lock, however, DBMS access to the file would be suspended. In order to prevent the DBMS from discovering this condition, the attacker could suspend the DBMS process temporarily until the tampering has been completed. An attacker with root privileges could also mark memory used by the DBMS as shared and tamper directly with memory.

Data Encryption. Different levels of encryption can be employed to protect database files, but they can ultimately be bypassed by an adversary with SA privileges. It is reasonable to assume that the SA would have the ability to decrypt any data that has been encrypted at the OS level. The SA would most likely not have the privileges to decrypt any internal database encryption. However, individual (value or record based encryption) is still subject to tampering since the metadata describing the encrypted values is still readable. Furthermore, column-level encryption values are decrypted when they are read into memory making it possible to map the decrypted values in memory back to the encrypted values in persistent storage.

5.1 Value Modification

The first category of file tampering action we consider is value modification. Value modification is logically similar to a SQL `UPDATE` command; this type of tampering results in extraneous data. Storage space and value encoding (see Section 3) are the main considerations when modifying a value.

If a modified value requires the same storage space as the original entry, no metadata needs to be updated. If the newly modified value requires less storage than the original, then metadata needs to be modified, and other values in the record may need to be shifted. For example, many DBMSes explicitly store string sizes on page – e.g., changing 'Hank' to 'Gus' requires metadata value with the size of the string to be changed from 4 to 3. Furthermore, if the modified value is not the last column in the record, all other columns must be shifted by one byte. Only the columns in the modified record need to be shifted; other records in the page can

remain as-is, leaving a gap (1 byte in our example). Shifting all other records in the page to close the gap would require all of the corresponding row directory addresses and relevant index pointers to be updated. If a value is modified to a value that requires more storage space, the old version of the record must be erased and the new version of the record must be appended to the table. These operations are discussed in the remainder of this section. Shifting the following records to accommodate a large value modification is not practical – unless the modified value happens to be in the last record on the page (and there is free space at the end of the page).

	(1) Starting Page	(2) Value Modification	(3) Record Addition	(4) Record Wiping
0	Checksum	Checksum'	Checksum'	Checksum'
	(A) 8100	8100	8050	8100
	(B) 2	2	3	1
	(C) 8150	8150	8150	8100
	8100	8100	8100	
	Free Space	Free Space	Free Space	Free Space
			44	
			2	
			4 Carl	
			7 Chicago	
8100	(D) 44	44	44	44
	(E) 2	2	2	2
	(F) 3 Bob	3 Bob	3 Bob	3 Bob
8150	(G) 6 Boston	6 Boston	6 Boston	6 Boston
	44	44	44	
	2	2	2	
	5 Alice	4 Andy	5 Alice	
8192	6 Austin	6 Austin n	6 Austin	0

Figure 5: Database file tampering examples.

Figure 5.2 shows an example of a value changed to a smaller size. Since ‘Andy’ is one byte smaller than ‘Alice’, the column size must be changed from 5 to 4. Furthermore, the name is not the last column so next column (‘Austin’) is shifted by one byte, which overwrites the ‘e’ at the end of ‘Alice’ and leaves an unused ‘n’ character from ‘Austin’.

5.2 Record Addition

The next file tampering action we consider is new record addition, which is logically similar to a SQL `INSERT` command. This type of file tampering results in extraneous data generated within the DBMS. When adding a record to a file, metadata in the row data, row directory, and page header must be considered along with the correct value encodings.

When a record is appended to an existing page, the structure of the record must match the proper active record structure for that DBMS. Section 3 discusses metadata that a DBMS uses to store records. For the DBMS to recognize a newly added record, a pointer must be appended to the page row directory. Finally, the free space pointer must be updated and the active record count (if used by the DBMS in question) must be incremented.

Figure 5.3 shows an example of the record (‘Carl’, ‘Chicago’) added to the page. Along with the values themselves, additional metadata is included in the row data. The size of each column, 4 and 7 bytes, is included, the column count, 2, and the row delimiter, 44. Next, a pointer, 8050, is added to the row directory, and the record count is updated to 3. Finally, the free space address is updated since the record was added to free space of the page.

5.3 Record Wiping

The final tampering action category we discuss is record wiping. Record wiping is logically similar to a SQL `DELETE` command, except that it fully erases the record. A proper SQL `DELETE` command will merely mark a record as deleted; record wiping explicitly overwrites the record to destroy the data, even from a forensic recovery tool. Record wiping erases data with no forensic trace as there is no indication that a record existed in a place where it was overwritten. Wiping a record from a file is essentially the reverse operation of adding a record to a file: the metadata in the row data, row directory, and page header must all be altered.

When a record is overwritten in a page, the entire record (including the metadata) is overwritten with the NULL ASCII character (a decimal value of 0). Next, the row directory pointer must also be overwritten in the same way. Finally, the free space pointer must be updated and the active record count (if used by the DBMS) must be decremented.

Figure 5.4 shows an example of the record (‘Alice’, ‘Austin’) erased from the page. Every byte used for the values and their metadata (column sizes, column count, and row delimiter) is overwritten with the decimal value 0. The row directory address for that row is erased and the row directory is defragmented. Finally, the record count is updated to 1.

Record Removal. Rather than explicitly overwriting a record, the record metadata could also be marked to mimic a SQL `DELETE`. We define such changes as a record removal (versus record wiping). We do not address record removal in this paper because such unlogged action can be detected by our previous work in [28] by comparing and flagging inconsistencies between DBMS storage forensic artifacts and the audit logs.

6 APPROACH OVERVIEW

Our goal in this paper is to eliminate a major security vulnerability stemming from file tampering; our solution is envisioned as a component of a comprehensive auditing system that employs database forensics. We have previously built a tool that detects malicious activity when database logging was disabled [28] by comparing forensic artifacts and database logs. That approach relied on forensic artifacts left by SQL commands and assumed no OS level file tampering. DBStorageAudit_{tor} finds inconsistencies that were done by direct file modification. Future work, such as recovering a time line of events or user attribution, would involve expanding upon the current components to the system.

The remainder of the paper describes our system to detect database file tampering, DBStorageAudit_{tor}, followed by an experimental evaluation in Section 11. Figure 6 provides an overview of DBStorageAudit_{tor}, which consists of four components: forensic extraction(A), index integrity verification(B), carved index sorting(C), and tampering detection(D).

The forensic processing component is based on the forensic tool DBCarver [31] described in Section 2. DBCarver retrieves from storage all table records (including deleted records), record metadata, index value-pointer pairs, and several additional storage artifacts. We discuss new functionality that was added to DBCarver for this paper in Section 7 (e.g., a page checksum extraction and comparison, a generalized approach to pointer deconstruction for several RDBMSes).

We first verify the integrity of indexes (discussed in Section 8) because indexes are used later to detect tampering of table data, so it is critical to verify index structure integrity. To achieve that, we evaluate the B-Tree in storage, consider corrupt data that

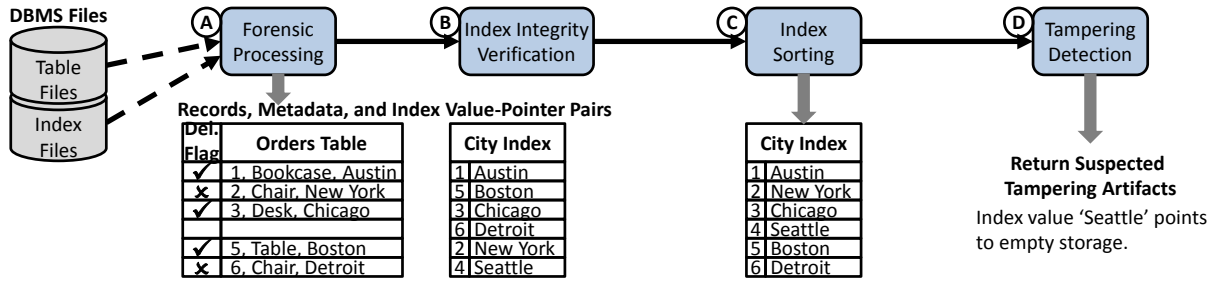


Figure 6: Architecture of the DBStorageAuditor.

matches B-Tree organization, and check for traces of an index rebuild (e.g., *REORG*, *VACUUM* – depending on a DBMS).

We cannot assume that index artifacts can be fully stored in RAM while matching index values to table records. Therefore, the carved index sorting component discussed in Section 9 pre-processes index artifacts to make DBStorageAuditor approach scalable. We approximately sort the index values based on their pointers which correspond to the physical location of records in a file and improves the runtime the matching process.

Finally the tampering detection component discussed in Section 10 detects cases of extraneous and erased data in storage. If a record and its artifacts can not be reconciled with index value-point pairs, such entries are flagged and returned to the user as suspected file tampering.

7 FORENSIC ANALYSIS

Our proposed analysis relies on an expanded version of DBCarver [31] to extract database storage artifacts that can not be queried using the DBMS. These artifacts include record metadata, deleted records, and index value-pointer pairs. In this section, we discuss the addition of a checksum comparison and generalized pointer deconstruction to DBCarver.

7.1 Checksum Comparison

In Section 3, we defined the checksum stored in the page header. Whenever data or metadata in a page is updated, either legitimately or through data tampering, the checksum must be updated accordingly. If the checksum is incorrect, the DBMS will recognize the page as corrupt. This will result in warnings as well as data loss ranging from page to the table or the entire database instance. Therefore, we can assert that if a checksum did not change between time T1 (previous inspection) and T2 (current inspection), then the page has not been modified and the records have not been exposed to tampering.

We implemented a dictionary of checksums taken from the DBMS pages that are to be evaluated by DBCarver (it is possible to inspect any subset of the DBMS for tampering signs – focusing only on data-sensitive tables). Our dictionary stores the checksum values, where the object identifier and page identifier (Section 3) were the key and the checksum was the value. The checksum dictionary should be stored off-site so it is not at risk of tampering.

If the checksum has changed for a given page, the entire page must be inspected and validated by DBCarver. If the checksum did not change for a page, only page metadata was necessary to reconstruct. The metadata is needed to avoid false-positives in Algorithm 2. Some DBMSes (e.g., Oracle, MySQL) allow the page checksum to be disabled. If the checksum is disabled or believed to have been disabled at some point, then a checksum comparison is unreliable and all data must be carved and inspected.

7.2 Index Carving and Pointer Deconstruction

DBStorageAuditor uses index value-pointer pairs to identify inconsistencies in DBMS storage. Therefore, the value-pointer pairs must be inspected. DBMSes do not allow indexes to be queried directly (i.e., indexes can not appear in the *FROM* clause) which is why we use DBCarver to retrieve index contents. However, the pointer parsing by DBCarver was limited and specific to each DBMS; we developed a generalized approach to pointer deconstruction allowing DBStorageAuditor to be compatible with any investigated RDBMS.

We performed an analysis of pointers for 7 commonly used RDBMSes. Table 1 lists these RDBMSes and summarizes our conclusions. We found that all of these DBMSes, except for MySQL, stored a PageID and a Slot#. By default, MySQL creates an indexed organized table (IOT) so the pointer deconstruction process is slightly different. We address index pointers for IOTs later in this section. The PageID refers to page identifier that is stored in table page header (Section 3). The Slot# refers to a records position within a page. SQLServer and Oracle both store a FileID, which refers to file in which the page is located. The DBMSes that do not include a FileID in the pointer, use a file-per-object storage architecture (i.e., each table and index are stored in different files). The FileID for these pointers is the ObjectID or it can be mapped back to the ObjectID if the object name is the file name. Thus, an index pointer can be deconstructed into a FileID, PageID, and Slot# to map a value back to a table record location. Index pointers are typically the same as the internal DBMS row identifier pseudo-column.

DBMS Version	FileID	PageID	Slot#
SQLServer	Yes	Yes	Yes
Oracle	Yes	Yes	Yes
ApacheDerby	No	Yes	Yes
PostgreSQL	No	Yes	Yes
Firebird	No	Yes	Yes
DB2	No	Yes	Yes
MySQL	No	Yes*	No

*The pointer references the second level of an IOT.

Table 1: Pointer Deconstruction.

Figure 7 demonstrates how index values are mapped back to the table records through our generalized pointer deconstruction. For each index value(A), the pointer stores a PageID(B) and Slot#(C). The pointer PageID(B) corresponds to the page identifier(D) in the table page header. The pointer Slot#(C) corresponds to the row directory address(E) in the table page. For example, the pointer for 'Austin' stores PageID = 8 and Slot# = 12. To find the record, the table page with identifier = 8 is found and the 12th row directory address is used to locate the record (68, 'Alice', 'Austin') within the page.

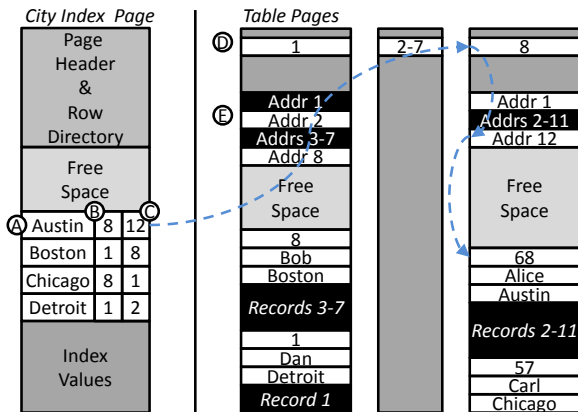


Figure 7: An example of mapping index values to a record.

Index Organized Tables. While MySQL was the only evaluated DBMS that created IOTs by default, IOTs are commonly used in other DBMSes under different names (e.g., IOT in Oracle, Included Columns in SQL Server) so we incorporated their pointer deconstruction. The pointer for a secondary index built on an IOT is made of a PageID that references a page one level above the IOT B-Tree leaf page, and the primary key value. The PageID for the IOT leaf page can then be retrieved from the pointer stored in the second level of the B-Tree. After performing this additional IOT B-Tree access, we can associate every secondary index value with a PageID and a primary key value, where the PageID references an IOT leaf page and the primary key value replaces the Slot#. Figure 8 illustrates how a secondary index value can be mapped back to an IOT record. We have the same index on *City* and the same records from Figure 7. However, the records are now stored in an IOT, and we now have a B-Tree page one level above the IOT leaf pages. The *City* index values(A) now store the PageID for IOT B-Tree page(B) and the primary key values(C) as the pointer. The IOT B-Tree page stores primary key values(F) and leaf PageIDs(G) as the pointer. For example, the pointer for 'Austin' stores PageID 20 and the primary key 68. This directs us to the IOT B-Tree page with PageID 20 and the value-pointer pair (57, 8). The IOT B-Tree pointer tells us 'Austin' is in the leaf page with PageID 8 and the primary key value 68.

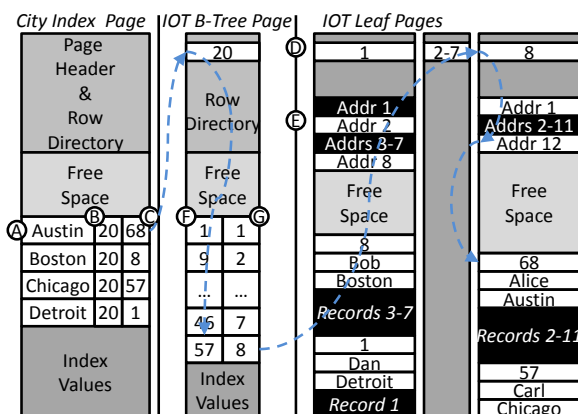


Figure 8: Mapping index values to an IOT record example.

8 VERIFYING INDEX INTEGRITY

It is plausible for an adversary to tamper with the relevant index values in an attempt to conceal evidence of file tampering. In this section, we address several types of index tampering, and how to detect such activity.

8.1 B-Tree Integrity Verification

If the attacker changes a value, adds a record, or wipes a record from a table, he may also perform a complimentary operation in the index. For example, 'Dan' was changed to 'Jane' in a table record could also be similarly modified in the index leaf node.

Interestingly, this type of activity creates inconsistencies in the index B-Tree that do not arise in the table. We consider the case where an index value is changed in-place and the case where index value was erased (and possibly reinserted into the correct position in the B-Tree). If the index value was changed in-place, it would appear out-of-order in the leaf of the B-Tree. If the index value was erased, it creates an uncharacteristic blank space between values within the leaf page, which *never* occurs naturally.

8.2 The Neighboring Value Problem

An index value may sometimes be altered without violating the correct ordering of the B-Tree. For example, in Figure 9 'Dan' is changed to 'Dog' preserving a correct value ordering of the Name index. This example shows how a table and an index can be altered without producing an inconsistency.

We build a function-based index that stores the hash value of column(s) to thwart tampering that involves neighboring range values. The values in hash-index will have a different ordering than the values in the secondary index so a neighboring value can occur in one, but not both. Figure 9 shows an example of how a hash index can be used to detect index tampering the involves neighboring values. In both the table and the Name index, the value was changed to 'Dog.' Changing the value in the Name index preserved the correct ordering. However, changing the value in the hash-index would result in an incorrect ordering since the values are organized differently. Function-based indexes are supported by many major DBMSes (e.g., IBM DB2, Oracle, and PostgreSQL); a computed column can be used for DBMSes that do not support function-based indexes (e.g., MySQL and Microsoft SQL Server).

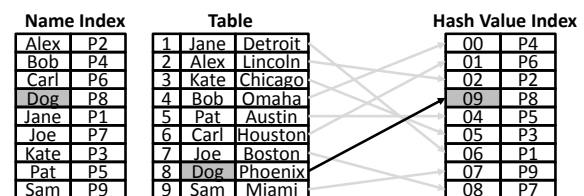


Figure 9: Preventing the neighboring value problem.

8.3 SQL Index Rebuild

Although we assume that the attacker does not have privileges to rebuild an index through SQL, the index may nevertheless be rebuilt as part of routine maintenance. If an index is rebuilt post tampering, the reconstruction of the index will eliminate any inconsistencies (extraneous or erased data) between the table and the index because indexes will be built anew using the *current* table state. However, when an object is rebuilt, a new object is created and artifacts (discarded pages) from the old object are left behind in storage. Many of the pages from the old index are likely to be overwritten, but some pages are going to persist in storage following the rebuild [30].

Pages left behind from an index rebuild can serve as separate evidence to detect tampering. The old index version (or the parts recovered) can be treated as a separate index (I_{t-1}) from the newly rebuilt version (I_t). While the old index version does not

contain a complete set of values due to having been partially overwritten, it can still be used to detect tampering. This would be applicable when auditing is not performed at regular intervals relative to the frequency of index rebuilds.

8.4 Manual Index Rebuild

In order to deceive DBStorage Auditor, an attacker would have to completely rewrite the entire index (or at least several different pages in it). While such operation is possible, performing it successfully poses several major challenges. We emphasize that typical security solutions are designed to greatly increase the level of difficulty to perform an attack, rather than create an absolute defense.

Section 5 discussed cached dirty page problem when physically modifying a page. Moreover, dirty index pages can introduce additional complications. First, a given index page is more likely to have a dirty version cached compared to a table page. An index page is not only modified when the indexed column is updated, but the index pointer must also be updated if an update causes a record to be written to a new location. Furthermore, index pages store significantly more values than table pages, increasing their chance to be modified. Second, as the index changes, the database may reorganize the B-Tree structure (e.g., page split). As parts of the index are rebuilt, pages are likely to be written to new locations in a file. We note that the physical order of a B-Tree does not reflect the logical order of the B-Tree. Third, the attacker may have to discover the physical location of other connected index pages (i.e., just finding the page with needed value is insufficient, several parts of the B-Tree would need to be reconstructed). Index leaf pages point to the next logical page in the B-Tree and sometimes to the previous page as well. This means that if a logically adjacent page is rebuilt and written to a new location, then a modified index page would need to reflect that change. Therefore, the attacker would need to be aware of all internal B-Tree structure changes to guarantee a successful manual index rebuild. Finally, if a function-based index storing a hash value exists, we assume that an SA would not have knowledge of this function. Therefore, inconsistencies would still arise in any attempts to manually rewrite the index.

9 INDEX SORTING

When tables and indexes are carved, the data is extracted based on the physical location within the files. Therefore, the relationship between the ordering of the carved table records compared to the index values is random, with a possible exception of a clustered index (it is common for a clustered index to be manually updated, such as PostgreSQL with VACUUM command). Assuming that the index can not be fully loaded into RAM, expensive random seeks must be performed to map index values to table records. In this section we propose a method to reorder the index to make the process of matching index values and table records scalable.

As demonstrated in Section 7, index pointers correspond to the physical position of the table records. Therefore, sorting the index values by the pointers produces the same ordering for index values and table records. Carved table records and index values are then read sequentially, similar to a merge join process.

For an index that is too large to fit into memory, sorting the index pointers can be a costly operation. If we assume that N table pages will be read into memory when detecting table tampering (Section 10), then index values need to be sorted across every N pages, but values do not need to be sorted within N pages.

We call each set of index values that belong in N table pages a **bucket**. We perform approximate sorting by re-ordering index values across buckets but not within buckets.

For each index bucket, we record the minimum and maximum table page identifier. If an index value is in the range of page identifiers for a bucket, the page identifier, slot number, and index value are stored in that bucket. When table pages are read for table tampering detection, the relevant bucket(s) are read into memory using the table page identifier and the index bucket minimum and maximum values.

Figure 10 shows an example of an index that is approximately sorted on the pointer. For each value in the index, there is a pointer that contains a PageID and a Slot#. We first create a set of buckets where each bucket contains 1000 PageIDs. We read the carved index data, and assign a value to the appropriate bucket using the pointer. For example, the first and second index values 'Alex' and 'Bob' belong in bucket #2 because their PageIDs, 2000 and 1002 are between the minimum and maximum PageID range for the bucket. We then store the PageID, Slot#, and Value in the bucket. 'Carl' has a PageID 5 so that value belongs in bucket #1. Bucket #2 demonstrates that PageIDs do not need to be sorted within the bucket. Furthermore, we see that PageID 2000 in bucket #2 has two values. This can occur as a result of legitimate SQL operations that create stale index values.

Carved Index			Approximately Sorted Index				
Value	Pointer		Bucket #	PageID Min/Max	PageID	Slot #	Value
Alex	2000	1	1	1 - 1000	5	33	Carl
Bob	1002	2	2	1001 - 2000	2000	1	Alex
Carl	5	33			1002	2	Bob
Dan	4400	12			1001	1	Joe
Jane	3050	20				2	Pat
Joe	1001	1	3	2001 - 3000	None		
Kate	1002	1	4	3001 - 4000	3050	20	Jane
Pat	1001	2	5	4001 - 5000	4400	12	Dan
Sam	2000	1					

Figure 10: An approximately sorted index example.

Our current implementation does not use the FileID even when it is stored in the pointer. We assume that each table is stored in a single file, and that the user has directed DBStorage Auditor to the relevant table and index files. DBMS-specific system tables would allow us to connect FileID to the information on target table and index files.

Index Organized Tables. Approximately sorting secondary indexes for index organized tables (IOT) is a slightly different process. When an IOT is used, the secondary index pointer is made up of a PageID that references a second level B-Tree page and the primary key value instead of a PageID that references the table and a Slot#. To sort the secondary index values, the second level BTree pages from the primary key is used to retrieve the table PageIDs for each value. Furthermore, the primary key value is now used in place of the Slot#.

The cost of approximate sorting is dependent on the amount of available memory. A bucket must fit into memory. Fewer buckets results in quicker bucket assignment for values, but buckets will be larger requiring more memory. In Section 11.2 we provide costs of approximately sorting an index.

10 DETECTING TABLE TAMPERING

In Section 5 we discussed how database files, specifically tables, are vulnerable to tampering. We propose using the validated indexes (Section 8) to verify the integrity of table records in storage. Earlier in this paper, we classified data tampering that

involves changing a value or adding records as extraneous data, and we classified data tampering that involves wiping records as erased data. In this section we present and discuss algorithms to detect both extraneous and erased data.

10.1 Extraneous Data Detection

Extraneous data is a record or a value that has been added to a table through file tampering. Since extraneous data is not added using the DBMS, it is not reflected in the indexes. Therefore, if a record does not have *any* corresponding index pointer, then the entire record is suspected of having been added through file tampering. Any table with a primary key can be tested because an index is automatically created for a primary key constraint. Similarly, if a table value does not match an index value with the corresponding pointer, then the value is assumed to have been modified through file tampering. This validation test does require that an index exist on the column(s). We use the carved data from Section 7 and an approximately sorted index (Section 9) that was not been subject to tampering (Section 8).

Algorithm 1 describes how to detect extraneous data. First, we read N table pages at a time for evaluation; we then scan the approximately sorted index buckets for the relevant table page identifiers and read the index pages from the relevant bucket(s). For every record in the N table pages, we find the corresponding index pointer. If an index pointer does not exist, this record is added to a list of likely extraneous data. If an index pointer does exist for a record, the indexed column is compared to the index value(s) for that pointer (there may be more than one index value per pointer for legitimate reasons). If the table value is not in the set of index values, then this value is added to a list of likely extraneous data. This is evidence of a value that has been changed. After all table pages have been read and all records evaluated, the resulting extraneous data list is returned to the user.

Algorithm 1 Extraneous Data Detection

```

1:  $Table \leftarrow$  carved table data: PageIDs, Slot #s, and Records.
2:  $N \leftarrow$  the number of table pages to be read.
3:  $SortedIndex \leftarrow$  the approximately sorted index (Section 9).
4:  $Flag \leftarrow$  an empty list to store extraneous data.
5: for each  $NPages \in Table$  do
6:    $MinPID \leftarrow$  the minimum page ID from  $NPages$ .
7:    $MaxPID \leftarrow$  the maximum page ID from  $NPages$ .
8:    $Indexes \leftarrow$  an empty list to store index pages.
9:   for each  $Bucket \in SortedIndex$  do
10:    if  $(MinPID \in Bucket) \vee (MaxPID \in Bucket) \vee$ 
       $(MinPID < Bucket \wedge MaxPID > Bucket)$  then
11:       $Indexes.append(Bucket)$ 
12:    for each  $Rec \in NPages$  do
13:       $RecPtr \leftarrow Rec.PageID.Slot\#$ 
14:      if  $RecPtr \in Indexes.PageID.Slot\#$  then
15:        if  $Rec.Val \notin Indexes.PageID.Slot\#.Vals$  then
16:           $Flag.append(['ModVal', RecPtr, Rec, Val])$ 
17:        else
18:           $Flag.append(['HiddenRecord', RecPtr, Rec])$ 
19: return  $Flag$ 

```

10.2 Erased Data Detection

Erased data is data explicitly wiped from table storage through file tampering. Deleted records are likely to be overwritten by new records over time as the DBMS runs. However, a deleted

record will never be overwritten by something that is not another record of the same structure. Therefore, if an index value points to an area in storage that does not contain a proper record (including metadata), then record wiping is suspected. We are not concerned with matching the specific index value since this is done in Algorithm 1, but rather that a pointer must reference an area in storage that resembles a record.

Algorithm 2 describes how to detect erased data. First, we read each bucket from the approximately sorted index. When a bucket is read, the table pages with the relevant page identifiers are also read. We iterate through each index value in the bucket. If the pointer for an index value does not match any record in the table pages, then the index value is appended to a list of erased data. After all index buckets have been evaluated, the list of erased data is returned to the user.

Algorithm 2 Erased Data Detection

```

1:  $Table \leftarrow$  carved table data: PageIDs, Slot #s, and Records.
2:  $SortedIndex \leftarrow$  the approximately sorted index (Section 9).
3:  $Flag \leftarrow$  an empty list to store erased data.
4: for each  $Bucket \in SortedIndex$  do
5:    $NPages \leftarrow$  pages from  $Table$  where  $PageID \in Bucket$ 
6:   for each  $IndexValue \in Bucket$  do
7:      $Ptr \leftarrow IndexValue.PageID.Slot\#$ 
8:     if  $Ptr \notin NPages$  then
9:        $Value \leftarrow$  the index value
10:       $Flag.append(['ErasedRecord', Ptr, Value])$ 
11: return  $Flag$ 

```

Adjacent Deleted Records. It is possible that multiple deleted records can exist adjacent to one another in a page. When this happens it is also possible the a single record could overwrite all of one record and part of another. For example, (1, 'Ed') and (2, 'Tom') are deleted records that are next to each other in storage. The inserted record (3, 'Karen') could overwrite all of (1, 'Ed') and part of (2, 'Tom'). This presents a problem because any old index value for (2, 'Tom') would now point to the middle of the inserted record, rather than to a full record. In this scenario, Algorithm 2 would return a false-positive for the index value from (2, 'Tom'). These false-positives can be eliminated by comparing these results to audit log entries. For example, if a delete command in the log could explain (2, 'Tom'), then this could be declared as not malicious. This functionality is not currently supported by DBStorageAuditor, and it would be explored in future work to achieve a more complete auditing system.

11 EXPERIMENTS

In this section, we present a set of experiments that evaluate the performance, accuracy, and limitations of DBStorageAuditor. Table 2 summarizes the experiments in this section.

MySQL 5.7, PostgreSQL 9.6, and Oracle 11g R2 DBMSes were used in these experiments. We believe these three RDBMSes are a good representative selection from the commonly used RDBMSes. Not only are they widely used commercial and open-source DBMSes, but they also represent the spectrum of different storage decisions across about ten DBMSes we have studied. For example, PostgreSQL does not support IOTs, Oracle offers an option to create IOTs, and MySQL automatically uses IOTs. The default page sizes for each DBMS were used: 8K for Oracle and PostgreSQL and 16K for MySQL. Data from the Star Schema Benchmark (SSBM) [16] was used to populate our DBMS instances. Table 3

#1	Forensic analysis (Sec 7) cost evaluation. DB files were carved at a rate of 1.2 MB/s. A checksum comparison can improve carving costs.
#2	Approximate sorting (Sec 9) cost evaluation. Fewer buckets improves runtime, but requires more memory.
#3	Algms 1 and 2 (Sec 10) cost evaluation. Both algorithms increase linearly with table size.
#4	DBStorageAuditor detection evaluation. Alg 1 detects an added record, Alg 1 detects a modified value only for an indexed column, and Alg 2 reconstructs erased data that was indexed.
#5	DBStorageAuditor detection limitations after an index rebuild (Sec 8). DBStorageAuditor can use the old version of an index depending on the DBMS.

Table 2: Summary of experiments.

can be used to reference table sizes used throughout this section. DBMS instances ran on servers with an Intel X3470 2.93 GHz processor and 8GB of RAM running Windows Server 2008 R2 Enterprise SP1 or CentOS 6.5.

Table	Scale	DB File Size(MB)	Values(M)
Lineorder	1	600	6
Lineorder	4	2400	24
Lineorder	14	8300	84
Supplier	1	<1	2K

Table 3: SSBM table sizes used through the experiments.

The different DBMS storage-altering operations that we are seeking to detect are discussed in Section 10. When modifying files, we re-calculated and updated the page checksum value for the PostgreSQL pages; in MySQL and Oracle we disabled the page checksum validation. Before modifying files, we first shutdown the DBMS instance.

11.1 Forensic Processing

The objective of this experiment is to evaluate the computational cost associated with the forensic processing component of DBStorageAuditor discussed in Section 7. In Part-A, we provide DBCarver runtimes against database files of various sizes from MySQL, Oracle, and PostgreSQL DBMSes. In Part-B, we repeat the same evaluation, further including a checksum re-computation.

Part-A. We created a series of database files for each DBMS to pass to DBCarver. We created three LINEORDER tables: Scale 1, 4, and 14. Each table was stored in a separate file. The PostgreSQL files were carved at an average rate of 1.0 MB/s, the MySQL files were carved at a rate of 1.2 MB/s, and the Oracle files were carved at a rate of 1.5 MB/s.

Part-B. We used the PostgreSQL LINEORDER Scale 4 table from Part-A to evaluate the checksum comparison we added to DBCarver. We modified pages that induced a checksum change for 1%, 5%, 10%, and 100% of the pages in the database file. The carving rate for each percent modification was 100% → 1MB/s, 10% → 9 MB/s, 5% → 18 MB/s, and 1% → 58 MB/s. The cost of forensic pre-processing is thus proportional to the number of modified pages rather than the total size of the DBMS storage.

11.2 Index Sorting

The objective of this experiment is to evaluate the costs associated with approximately sorting the index values on the pointers. The output produced by the forensic analysis is similar for all DBMSes

so this component of DBStorageAuditor is not tested for DBMS-specific features. In Part-A, we vary the size of bucket; in Part-B, we vary the size of the indexes.

Part-A. To evaluate approximate sorting with respect to bucket size, we used the carved output from PostgreSQL database files containing a LINEORDER Scale 4 table, a secondary index on LO_Revenue, and a secondary index on LO_Orderdate. Table 4 summarizes the performance results. As the number buckets decreases the time to sort the data decreases. However, a bucket must fit into memory, so increasing of bucket sizes is limited by available RAM.

Bucket Size (Pages)	Bucket Count	Orderdate (sec)	Revenue (sec)
5,000	63	1366	1380
10,000	32	1121	1131
50,000	7	932	945
100,000	4	909	926
200,000	2	903	918

Table 4: Index sorting costs with varying bucket sizes.

Part-B. To evaluate approximate sorting with respect to the size of an index, we used the carved output from PostgreSQL database files containing LINEORDER Scale 1, 4, and 14 tables and a secondary index on LO_Revenue for each table. Table 5 summarizes the results. If the bucket size is increased proportionally for the table size, the approximate sorting cost increases linearly.

Bucket Size (Pages)	Index sorting time (sec)		
	Scale 1	Scale 4	Scale 14
10,000	239	1131	6193
50,000	231	945	3797
100,000	n/a*	926	3486
200,000	n/a*	918	3357

*Bucket size is larger than the table.

Table 5: Approximate sorting costs for varying table sizes.

11.3 Tampering Detection Costs

The objective of this experiment is to evaluate the costs associated with of Algorithms 1 and 2. For this experiment we used the LINEORDER Scale 4 table. We used one index on the LO_Revenue and multiple indexes on the LO_Revenue and LO_Orderdate. We approximately sorted the index using buckets with 50K pages.

Part-A: Algorithm 1. To evaluate the costs associated with Algorithm 1, we used the output from two different secondary indexes (LO_Revenue and LO_Orderdate) on LINEORDER Scale 4 and one secondary index (LO_Revenue) on LINEORDER Scale 14. Table 6 summarizes the runtime results. The runtime for Algorithm 1 was the same for LO_Revenue and LO_Orderdate on LINEORDER Scale 4, and the cost increased linearly for LO_Revenue on LINEORDER Scale 14.

Table	Index	Part-A (sec)	Part-B (sec)
Scale 4	LO_Revenue	966	503
Scale 4	LO_Orderdate	961	476
Scale 14	LO_Revenue	3482	1773

Table 6: Algorithm 1 and 2 runtimes.

Part-B: Algorithm 2. We used the same tables in indexes from Part-A of this experiment to evaluate the costs associated with Algorithm 2. Table 6 summarizes the runtime results. Similar to Algorithm 1, the cost for Algorithm 2 was nearly the same for LO_Revenue and LO_Orderdate on LINEORDER Scale 4, and the cost increased linearly for LO_Revenue on LINEORDER Scale 14.

11.4 Detection Capabilities

The objective of this experiment is to demonstrate the file tampering activity that DBStorageAuditor is capable of detecting. For each part in this experiment, we simulate one defined type of malicious activity and explain how it was detected. We manually add records to the database file (Part-A), change values in the database file (Part-B), and erase records from the database file (Part-C). We present results only for PostgreSQL because we our results for Oracle and MySQL were very similar.

Setup. We created a LINEORDER Scale 4 table for a PostgreSQL DBMS. An index existed on the primary key (LO_Orderkey, LO_Linenum) and we created a secondary index for LO_Revenue and LO_Orderdate.

We also created a function-based index on LO_Revenue that used the 32-bit version of the MurmurHash2 hash function.

Part-A. We manually added 5 records (shown in Figure 11) to the file containing the LINEORDER table. We added a record to five different pages (with PageIDs 11, 12, 13, 14, and 15). Existing primary key values were included in each of the five records. For each of these records, all of the data was the same as the existing records with the same primary key except we used LO_Supkey -5 and LO_Revenue -100000.

Primary Key	LO_Supkey = -5	LO_Revenue = -100000
① 1011 108733 7417	-5 19960319 '3-MEDIUM' 0 49	7352695 20527439 10 -100000 90033 0 19960529 'AIR'
② 40011 38143 210370	-5 19931228 '1-URGENT' 0 26	3328936 3362225 0 -100000 76821 1 19940113 'RAIL'
③ 120011 2303 391486	-5 19970718 '4-NOT SPECI' 0 8	1261976 17693973 1 -100000 94648 1 19971011 'SHIP'
④ 1000011 102599 383999	-5 19941106 '3-MEDIUM' 0 14	2916172 2995491 4 -100000 124978 7 19950117 'SHIP'
⑤ 2000011 85157 130108	-5 19960903 '1-URGENT' 0 21	2390010 2413431 1 -100000 68286 2 19961005 'REG AIR'

Figure 11: Records added to the LINEORDER file.

The addition of these five records produced several interesting outcomes. First, these records bypassed the primary key constraint since they contained primary key values that previously existed in the table. The DBMS only checks constraints when executing API-based load commands, and it does not retroactively check the table for constraint violations. Adding the record to the file bypasses all official channels and is thus never checked for constraint violations. Second, these records also bypassed referential integrity since the LINEORDER table references the SUPPLIER table, and LO_Supkey -5 did not exist in the SUPPLIER. Similar to the primary key violation, the constraint violation was never caught by the DBMS. Finally, table access for the same query could produce different results because the indexes were not updated after we added these five records. For example, the two versions of the following query returns different results:

- Query 1 → 34600180980

```
SELECT SUM(LO_Revenue) FROM Lineorder
WHERE LO_Orderdate = 19960319;
```
- Query 2 → 34600180980 - 100000

```
set enable_seqscan=true;
SELECT SUM(LO_Revenue) FROM Lineorder
WHERE LO_Orderdate = 19960319;
```

Query 1 uses the LO_Orderdate index to access the table while Query 2 uses a full table scan. Record #1 from Figure 11 was included in Query 2, but it was not included in Query 1.

Algorithm 1 successfully detected the fact that five new records do not have corresponding pointers in the primary key index, the two secondary indexes, and in the function-based index. Problem was flagged by a False value for the line 14 If condition resulting in the malicious records being added to the list of invalid data at line 18. Each existing index serves as an additional validation to detect table tampering – and the function-based makes sure that small incremental changes are not possible.

Part-B. Next, we changed LO_Revenue for all 41 records where the LO_Custkey 4321 and LO_Orderdate between 19930101 and 19931231. To simulate a neighboring value problem (a small change that does not violate index ordering), we changed the record with LO_Custkey 4321 and LO_Revenue 3271986 to 3271987 in both the table and the LO_Revenue index. For all other records we subtracted 100000 from LO_Revenue in the table.

Algorithm 1 reported that 40 records had an inconsistent value based on the LO_Revenue index and 41 records had an inconsistent value based on the function-based index on LO_Revenue. The difference of the one additional record was due to the neighboring value attack which regular index may fail to detect. These values were detected by a False value for the line 15 If condition resulting in the malicious values being added to the list of invalid data at line 16. We can conclude that the primary key and LO_Orderdate columns were not tampered with for these and all records since they were not included in the invalid data. However, we can not make any conclusion if any other of the non-indexed columns for these or any records were tampered.

Part-C. Next, we erased all 3085 records with the LO_Supkey 123 from the file. For data erasure, we explicitly overwrote the records and their metadata with the NULL ASCII character.

Algorithm 2 returned that primary key index, the two secondary indexes, and the function-based index each had 3085 values that did not point to a valid record structure. These were detected by a True value for the line 8 condition in Algorithm 2, resulting in malicious data being added to the list of invalid data at line 10. By combining the values for each pointer we reconstructed partial records containing the index columns to explain the missing data. However, the data for the non-indexed columns was unable to be reconstructed since it was not indexed.

11.5 Long-Term Detection

The objective of this experiment is to evaluate the artifacts produced by an index rebuild that can be used by DBStorageAuditor. We evaluate a different DBMS for each part of this experiment: Oracle in Part-A, MySQL in Part-B, and PostgreSQL in Part-C.

We performed the following steps for each DBMS. After each step, we copied the database file for analysis. Table 7 summarizes the results.

- T_0 : Started with the Supplier Scale1 (2K records) table and a secondary index on S_Name.
- T_1 : Erased/wiped all 829 records where S_Region equaled 'ASIA' or 'EUROPE'.
- T_2 : Rebuild the index. Each DBMS used a different index rebuild command:
 - Oracle: ALTER INDEX Supp_Name REBUILD ONLINE
 - MySQL: DROP and CREATE commands
 - PSQL: REINDEX TABLE Supplier

Part-A: Oracle. The index contained 1 root page and 9 leaf pages after creation at T_0 . No changes were made to the index after the table records were erased at T_1 . After the index rebuild at T_2 , the new index contained 1 root page and 5 leaf pages. All of

DBMS	T_0 (pgs)	T_1	T_2
Oracle	1 root, 9 leaf	no change	All index pages from the old index remained in DB storage.
MySQL	1 root, 5 leaf	no change	2 leaf pages from the old index remained in DB storage.
PSQL	1 root , 10 leaf	no change	None of the old index remained in DB storage.

Table 7: Index rebuild summary.

the pages from the original version at T_0 remained in the database file. The DBMS assigned a new ObjectID to the new version of the index so index pages between versions were easily distinguished. Since the entire version of index was found, it could be used by DBStorageAuditor. The old version of the index still contained pointers to the erased records, whereas the new version only contained pointers to active records in the table.

Part-B: MySQL. The index contained 1 root page and 5 leaf pages after creation at T_0 . No changes were made to the index after the table records were erased at T_1 . After the index rebuild at T_2 , the new index contained 1 root page and 3 leaf pages. 2 out of the 5 leaf pages from the original index remained in database storage. This demonstrates that the DBMS immediately reclaimed the pages from the dropped index. Since the new index version used less storage space, 2 pages from the old version remained in the file. In this scenario, a B-Tree could not be fully validated with only 2 leaf pages making them less useful as evidence for DBStorageAuditor. It is likely that copies of the index could be carved from a disk image due to activity such as writes that do not occur in place and paging files. DBStorageAuditor does not currently reconstruct entire B-Tree indexes from disk images. Future work will seek to reconstruct objects from disk images, which requires multiple versions of pages to be considered.

Part-C: PostgreSQL. The index contained 1 root page and 10 leaf pages after creation at T_0 . No changes were made to the index after the table records were erased at T_1 . After the index rebuild at T_2 , the new index contained 1 root page and 6 leaf pages. The new version of the index was assigned a new ObjectID and a separate file. All pages belonging to the old version of the index were disassociated with its file, and this storage was reclaimed by the file system. In this scenario, DBStorageAuditor can no longer detect that the records were erased. As discussed in Part-B, it is likely that the index could be carved from a disk image. This will be explored in future work since a logical timeline would need to be recreated to account for multiple page versions.

12 CONCLUSION

Database file tampering can be used to perform malicious operations while bypassing database security mechanisms (logging and access control) and constraints. We presented and evaluated DBStorageAuditor component that detects database file tampering. Our approach relies on a forensic inspection of database storage and identifies inconsistencies between tables and indexes.

Future work plans to expand upon this paper and work from [28] to create a complete database auditing framework. This future work would include creating a timeline of events and user attribution of storage artifacts. Our auditing framework relies on inherent characteristics of database storage that users, including privileged users, are incapable of controlling.

ACKNOWLEDGMENTS

This work was partially funded by the US National Science Foundation Grant CNF-1656268.

REFERENCES

- [1] Brian Carrier. 2011. The sleuth kit. <http://www.sleuthkit.org> (2011).
- [2] Scott A. Crosby and Dan S. Wallach. 2009. Efficient Data Structures for Tamper-evident Logging. In *Proceedings of the 18th Conference on USENIX Security Symposium (SSYM'09)*. USENIX Association, Berkeley, CA, USA, 317–334. <http://dl.acm.org/citation.cfm?id=1855768.1855788>
- [3] Daniel Fabbri, Ravi Ramamurthy, and Raghav Kaushik. 2013. SELECT triggers for data auditing. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*. IEEE, 1141–1152.
- [4] Lee Garber. 2001. Encase: A case study in computer-forensic technology. *IEEE Computer Magazine* January (2001).
- [5] Simson Garfinkel. 2007. Anti-forensics: Techniques, detection and countermeasures. In *2nd International Conference on i-Warfare and Security*, Vol. 20087. Citeseer, 77–84.
- [6] Simson L Garfinkel. 2007. Carving contiguous and fragmented files with fast object validation. *digital investigation* 4 (2007), 2–12.
- [7] Michael T Goodrich, Mikhail J Atallah, and Roberto Tamassia. 2005. Indexing information for data forensics. In *ACNS*, Vol. 5. Springer, Citeseer, 206–221.
- [8] Ryan Harris. 2006. Arriving at an anti-forensics consensus: Examining how to define and control the anti-forensics problem. *digital investigation* 3 (2006), 44–49.
- [9] IBM. 2017. IBM Security Guardium Express Activity Monitor for Databases. <http://www-03.ibm.com/software/products/en/ibm-security-guardium-express-activity-monitor-for-databases>. (2017).
- [10] Gary C Kessler. 2007. Anti-forensics and the digital investigator. In *Australian Digital Forensics Conference*. Citeseer, 1.
- [11] Gene H Kim and Eugene H Spafford. 1994. The design and implementation of tripwire: A file system integrity checker. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*. ACM, 18–29.
- [12] Lianzhong Liu and Qiang Huang. 2009. A framework for database auditing. In *Computer Sciences and Convergence Information Technology, 2009. ICCIT'09. Fourth International Conference on*. IEEE, 982–986.
- [13] ManageEngine. [n. d.]. EventLog Analyzer. <https://www.manageengine.com/products/eventlog/>. ([n. d.]).
- [14] Charles Martel, Glen Nuckolls, Premkumar Devanbu, Michael Gertz, April Kwong, and Stuart G Stubblebine. 2004. A general model for authenticated data structures. *Algorithmica* 39, 1 (2004), 21–41.
- [15] OfficeRecovery. [n. d.]. Recovery for MySQL. <http://www.officerecovery.com/mysql/>. ([n. d.]).
- [16] Patrick O.Neil, Elizabeth O.Neil, Xuedong Chen, and Stephen Revilak. 2009. The star schema benchmark and augmented fact table indexing. In *Performance evaluation and benchmarking*. Springer, 237–252.
- [17] Kyriacos E Pavlou and Richard T Snodgrass. 2008. Forensic analysis of database tampering. *ACM Transactions on Database Systems (TODS)* 33, 4 (2008), 30.
- [18] Jon M Peha. 1999. Electronic commerce with verifiable audit trails. In *Proceedings of ISOC*. Citeseer.
- [19] Percona. [n. d.]. Percona Data Recovery Tool for InnoDB. <https://launchpad.net/percona-data-recovery-tool-for-innodb>. ([n. d.]).
- [20] Stellar Phoenix. [n. d.]. DB2 Recovery Software. <http://www.stellarinfo.com/database-recovery/db2-recovery.php>. ([n. d.]).
- [21] Golden G Richard III and Vassil Roussev. 2005. Scalpel: A Frugal, High Performance File Carver. In *DFRWS*. Citeseer.
- [22] Christian S. J. Peron and Michael Legary. 2017. Digital Anti-Forensics: Emerging trends in data transformation techniques. (09 2017).
- [23] Bruce Schneier and John Kelsey. 1999. Secure audit logs to support computer forensics. *ACM Transactions on Information and System Security (TISSEC)* 2, 2 (1999), 159–176.
- [24] Arunesh Sinha, Limin Jia, Paul England, and Jacob R Lorch. 2014. Continuous tamper-proof logging using TPM 2.0. In *International Conference on Trust and Trustworthy Computing*. Springer, Springer, 19–36.
- [25] Richard T Snodgrass, Shilong Stanley Yao, and Christian Collberg. 2004. Tamper detection in audit logs. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30, VLDB Endowment*, 504–515.
- [26] Patrick Stahlberg, Gerome Miklau, and Brian Neil Levine. 2007. Threats to privacy in the forensic analysis of database systems. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. ACM, Citeseer, 91–102.
- [27] Roberto Tamassia. 2003. Authenticated data structures. In *ESA*, Vol. 2832. Springer, Springer, 2–5.
- [28] James Wagner, Alexander Rasin, Boris Glavic, Karen Heart, Jacob Furst, Lucas Bressan, and Jonathan Grier. 2017. Carving database storage to detect and trace security breaches. *Digital Investigation* 22 (2017), S127–S136.
- [29] James Wagner, Alexander Rasin, and Jonathan Grier. 2015. Database forensic analysis through internal structure carving. *Digital Investigation* 14 (2015), S106–S115.
- [30] James Wagner, Alexander Rasin, and Jonathan Grier. 2016. Database image content explorer: Carving data that does not officially exist. *Digital Investigation* 18 (2016), S97–S107.
- [31] James Wagner, Alexander Rasin, Tanu Malik, Karen Heart, Hugo Jehle, and Jonathan Grier. 2017. Database forensic analysis with DBCarver. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research*.

User-guided Repairing of Inconsistent Knowledge Bases

Abdallah Arioua*

University Lyon 1 & CNRS Liris
abdallah.arioua@univ-lyon1.fr

Angela Bonifati†

University Lyon 1 & CNRS Liris
angela.bonifati@univ-lyon1.fr

ABSTRACT

Repairing techniques for relational databases have leveraged integrity constraints to detect and then resolve errors in the data. User guidance has started to be employed in this setting to avoid a prohibitory exploration of the search space of solutions. In this paper, we present a user-guided repairing technique for Knowledge Bases (KBs) enabling updates suggested by the users to resolve errors. KBs exhibit more expressive constraints with respect to relational tables, such as tuple-generating dependencies (TGDs) and negative rules (a form of denial constraints). We consider TGDs and a notable subset of denial constraints, named contradiction detecting dependencies (CDDs). We propose user-guided polynomial-delay algorithms that ensure the repairing of the KB in the extreme cases of interaction among these two classes of constraints. To the best of our knowledge, such interaction is so far unexplored even in repairing methods for relational data. We prove the correctness of our algorithms and study their feasibility in practical settings. We conduct an extensive experimental study on synthetically generated KBs and a real-world inconsistent KB equipped with TGDs and CDDs. We show the practicality of our proposed interactive strategies by measuring the actual delay time and the number of questions required in our interactive framework.

1 INTRODUCTION

Integrity constraints have been used in relational databases to detect inconsistencies and thus repair error-prone data within tables. Notable classes of these constraints are represented by functional dependencies (FDs) and conditional functional dependencies (CFDs) that are both table-level constraints as they express conditions without or with predicates on entire relations. Denial constraints (DCs) [7] are more general first-order formulas that encompass FDs and CFDs and strike a balance between expressiveness and complexity. Denial constraints are, however, difficult to understand for end users and their intractability and prohibitive search space make them unattractive for repairing algorithms [3].

In this paper, we focus on a subset of denial constraints, which we call contradiction-detecting dependencies (CDDs) capturing contradictions in the data. CDDs correspond to denial constraints restricted to equality predicates in their respective bodies. They are used mainly to capture contradictions and disjointness between relations. They differ from other subfamilies of DCs such as keys, functional dependencies and equality-generating dependencies. Knowledge Bases (KBs) typically rely on the interaction of CDDs (also known as negative rules or negative constraints) with tuple-generating dependencies (also called existential rules) [5, 11]. The following example underlines the importance of CDDs.

*Supported by PALSE Impulsion.

†Partially supported by CNRS Mastodons MedClean and PALSE Impulsion.

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

Example 1.1. Figure 1 (a) shows our running example. Let \mathcal{F} contain the set of facts of a KB describing the prescriptions of patients at a hospital and Σ_C the set of CDDs. *Aspirin* is prescribed to *John* who is allergic to it, whereas *Mike* has an allergy against *Penicillin*. The CDD in Σ_C dictates that prescribing a drug to a person who has an allergy against it leads to a contradiction.

Several approaches to repair KBs exist, such as deletion-based repairing, which amounts to remove the inconsistencies in order to satisfy the constraints. However, the generated repairs are not compatible, as a consequence, choosing among them is not feasible for end-users, as shown by the following example.

Example 1.2. Following the deletion-based repairing approaches, one can either remove *prescribed(Aspirin, John)* or *hasAllergy(John, Aspirin)* as either one of them is false according to the CDD. The first one gives us the repair \mathcal{F}_1 that conveys the information that *Aspirin* is prescribed to *John*. Conversely, the second repair \mathcal{F}_2 would lead us to conclude that *John* has an allergy against *Aspirin*. Moreover, none of the above repairs preserves as much information as possible. The information about *John* having an allergy that could be against *Aspirin* or any other drugs is indeed lost in \mathcal{F}_1 whereas the information that *Aspirin* is prescribed to someone which could be *John* or someone else is also lost in \mathcal{F}_2 . This example also shows the impossibility for an end-user of making a choice between these two repairs.

An alternative to deletion-based repairing is given by update-based repairing [24, 28], which inspired our work. In update-based repairing, atomic values can be modified instead of removing entire facts from the knowledge base.

Example 1.3. By applying an update-based repairing to the above inconsistent knowledge base, we obtain the set of facts \mathcal{F}_3 (in which X_1 is a labeled null referring to an unknown allergy). Another possible repair is that *John* has an allergy against *Penicillin* rather than *Aspirin*. Or, *Penicillin* is prescribed to *John* rather than *Aspirin*. Clearly, all these update-based repairs preserve more facts than the deletion-based ones illustrated above.

Although being beneficial, update-based repairing suffers from the problem of choosing the positions to modify and the value to use in repairing. For instance, do we need to change *Aspirin* to *Penicillin* in *hasAllergy(John, Aspirin)* or *Aspirin* to a labeled null? Clearly, user intervention is compulsory in such a case in order to reach a repair that meets the user's requirements and his expertise about the domain.

The problem becomes more complex when CDDs and TGDs are considered as shown in Figure 1 (b). Besides the fact that \mathcal{F} already contains inconsistencies as illustrated in Figure 1 (a), we consider Σ_T and the new introduced atoms in \mathcal{F} and a new CDD in Σ_C' . In this KB, another inconsistency is raised due to the interaction between TGDs and CDDs corresponding to the fact that *John* was prescribed incompatible drugs, i.e. *Aspirin* and *Nsaids*. Such a contradiction can only be discovered after applying the TGD that results in deducing the fact *John* is prescribed *Nsaids*, because he has *Migraine* pain and *Nsaid* is a painkiller. In such a

$$\begin{aligned} \mathcal{F} &= \{prescribed(Aspirin, John), hasAllergy(John, Aspirin), hasAllergy(Mike, Penicillin)\} \\ \Sigma_C &= \{prescribed(X, Y), hasAllergy(Y, X) \rightarrow \perp\} \\ \mathcal{F}_1 &= \{prescribed(Aspirin, John), hasAllergy(Mike, Penicillin)\} \\ \mathcal{F}_2 &= \{hasAllergy(John, Aspirin), hasAllergy(Mike, Penicillin)\} \\ \mathcal{F}_3 &= \{prescribed(Aspirin, John), hasAllergy(John, X_1), hasAllergy(Mike, Penicillin)\} \end{aligned}$$

(a) An inconsistent knowledge base with only CDDs. \mathcal{F}_i are repairs.

$$\begin{aligned} \mathcal{F}' &= \mathcal{F} \cup \{hasPain(John, Migraine), isPainKillerFor(Nsaids, Migraine), incompatible(Aspirin, Nsaids)\} \\ \Sigma_T &= \{isPainKillerFor(X, Y), hasPain(Z, Y) \rightarrow prescribed(X, Z)\} \\ \Sigma_{C'} &= \Sigma_C \cup \{prescribed(X, Z), prescribed(X, Y), incompatible(Y, Z) \rightarrow \perp\} \end{aligned}$$

(b) An inconsistent knowledge base with CDDs and TGDs.

Figure 1: Examples on tuple-based repairing and update-based repairing.

case, after applying the rule in Σ_T , a new inconsistency has been introduced. Hence, the choice of which inconsistency to handle first and which atom to update is crucial. For instance, updating the atom $prescribed(Aspirin, John)$ will resolve automatically the new inconsistency without updating other atoms, whereas updating the atom $prescribed(Nsaids, John)$ will not. In addition, propagating back the changed positions in $prescribed(Nsaids, John)$ should be done in order to establish consistency.

In this paper, we present a user-guided update-based repairing framework that is capable of solving contradictions triggered by CDDs. We study the interaction of such rules with more classical tuple-generating dependencies (or, existential rules) in KB reasoning. The study of DCs in a relational setting has been extensively conducted in the literature as witnessed by several papers in the area [13, 24, 28]. We defer the discussion of the differences between our work and previous work to the next subsection. In this paper, we focus on the following problem statement, which substantially deviates from the objectives of previous work.

(URP) *Given a KB equipped with a set of TGDs and CDDs, the User-guided Repairing Problem is to compute, by means of user's update fixes, an error-free KB by addressing two main challenges: (i) minimizing user interactions and (ii) accounting for the interplay of TGDs and CDDs. If the user is an oracle, then the repair of the KB is also a u-repair, i.e. a repair with a minimal (w.r.t \subseteq) set of update fixes.*

Contributions. The main contributions of our paper are as follows:

(1) Update-based repairing: We introduce contradiction detection dependencies (CDDs) and we formalize update-based repairing in the presence of both CDDs and TGDs. We prove that repairability is guaranteed and can be checked in polynomial time.

(2) Update-based repairing with user interaction: We define an interactive framework letting the user repair the knowledge base and meet his requirements. We prove two interesting properties: (i) *soundness*, i.e. we show that the framework is sound, which means that for every dialogue with a user we can reach a consistent state of the knowledge base; (ii) *soundness w.r.t. an oracle*, i.e. we assume that the interaction is done with an oracle that has a specific repair in mind and we prove that the output of the dialogue with an oracle is exactly the repair of the oracle. We show that the dialogue algorithm has a polynomial delay in generating questions.¹

We present an extensive experimental study, devoted to confirm the polynomiality of delay time and showing the feasibility of our interactive approach in terms of number of questions and average number of conflicts per question in the knowledge bases. In our assessment, we contrast the two cases of only CDDs and

CDDs alongside with TGDs and we study the impact of the chase algorithm on the proposed interactive strategies in both cases.

Paper organization. The paper is organized as follows. In Section 2, we introduce the basic notions and definitions. In Section 3, we introduce the update-based repairing framework and the repairability of KBs. In Section 4, we formalize user intervention by means of the notion of inquiry and we prove the soundness and termination of an inquiry engaging the end-user. We also discuss the case in which the user is an oracle and prove that the inquiry has a polynomial delay time. In Section 5, we introduce different interactive strategies with the user. In Section 6, we present our experimental assessment. Finally, Section 7 concludes the paper.

1.1 Related Work

Rule-Based Repairing. Logical data cleaning has leveraged reasoning over more or less sophisticated classes of declarative dependencies [9, 14] in order to detect and repair error-prone values and tuples. A plethora of data quality constraints have been introduced to this purpose, ranging from classical functional dependencies and their approximate variants for relational tables [9, 27] to their counterparts in graph databases [17]. Denial constraints [7] are first-order formulas more expressive than functional dependencies and conditional functional dependencies. Comprehensive classes of graph constraints, including the expressive graph entity dependencies (encompassing denial constraints) have been presented in [16]. The detection problem [17] for these constraints consists in checking whether a given database (or a graph) contains no violations of the input set of constraints. While [16] focuses on the detection problem along with satisfiability and implication among constraints, our goal in this paper is to compute an update-based repair for a knowledge base with an interactive exploration of the search space of solutions. Our contradiction detecting dependencies are a subset of denial constraints, limited to equality as built-in predicate. While denial constraints have been already employed as data quality rules in the relational setting [13], their use in the realm of knowledge bases characterized by the schema-less nature of data and their combination with other KB constraints, such as TGDs, is not explored. CLAMS [18] investigates the use of DCs in data lakes, including RDF KBs, but it does not consider the TGDs and their interactions with DCs. With that being said, our approach goes with the same line of [13, 18] in reinforcing and endorsing a holistic view of repairing for knowledge bases by compiling the information coming from multiple violations in a structure called the Conflicts Hypergraph [13, 24]. Repairing a database [1, 7, 28] according to a set of constraints corresponds to bringing the database to its legal state, in which all the constraints are satisfied. Since many possible repairs for a given database may exist, one would tend to prefer the (minimal) one, which entails less modifications of the original database. Various notions

¹The delay between the asked questions is bounded by a polynomial.

of repairs have been used and many approaches have used insert and delete operations on the original database to make it reach its consistent state with respect to a given class of constraints. Such approaches may exhibit drawbacks in that the granularity of the operations performing the repairs is too coarse. Indeed, deletions and insertions are typically executed at the tuple level for relations, thus leading to discard possibly error-free values. Our work has been inspired by update-based repairing proposed in [10, 28] to allow value replacement on positional attributes in relational tuples. While [28] focused on consistent query answering for update repairs aiming at finding the answers of a query in the intersection of all possible repairs, our intent is to exploit user interactions in the update-based repairing process of an entire knowledge base. Repairing by value modification with functional dependencies and inclusion dependencies has been tackled in [10] with the aim of building minimal-cost repairs. Their algorithms are not directly applicable to KBs due the inherent difference of expressiveness of the constraints and the consequent interaction between tuple-generating dependencies and the CDDs. In addition, user intervention has not been considered in [10].

User-guided data cleaning. A fruitful line of work has led to the design of several data cleaning tools, such as Llunatic [19], GDR [29], Katara [14], Dance [4] and Falcon [21].

GDR [29] considers user-guided relational data repairing. CFDs are used to generate candidate updates for the tuples that are violating them. The user is presented with groups of updates and her feedback is fed into an active learning process that decides about the correctness of updates without user involvement. The convergence of updates in our method is ensured by the chase algorithm involving CDDs and TGDs on KBs.

Llunatic [19] is mapping and cleaning tool accepting user suggestions during the chase procedure with EGDs on relational instances. Llunatic also explores the interaction among several classes of constraints such as FDs, CFDs, editing rules and TGDs. To the best of our knowledge Llunatic cannot be directly applied to knowledge bases with constraints such as CDDs and TGDs.²

Falcon [21] relies on a set of SQL update queries instead of a set of input logical constraints to entail the repair of a relational database. A set of SQLU queries is inferred starting from one triggering input tuple-based update proposed by the non-expert user. Our approach is based on rule-based repairing of knowledge bases and on a tight interaction with the domain expert to perform data curation, not considered in the above system.

Dance [4] introduces a user-driven cleaning approach for relational tuples, by considering constraints similar to classical EGDs and TGDs. Dance proposes a set of suspicious tuples whose update can contribute to constraint resolution. However, neither they consider DCs or subset thereof employed in our framework nor they leverage their interaction with TGDs as in our approach.

Katara [14] is orthogonal to our work in that it leverages knowledge bases and guidance from crowdsourcing to fix the errors in RDBMS. Because of that, input KBs are assumed to be well curated, as opposed to the assumption undertaken in our paper.

2 PRELIMINARIES

In this section, we briefly recap the notions needed in our framework, namely the definition of a knowledge base and the corresponding constraints, along with the definition of conflicts.

²Benchmarks on LUBM in [6] have been performed on a relational representation of the LUBM ontology with vertical partitioning.

Constraints and KBs. A *tuple-generating dependency* (abbreviated TGD) is of the form $R : \forall \mathbf{x} \forall \mathbf{y} B(\mathbf{x}, \mathbf{y}) \rightarrow \exists \mathbf{z} H(\mathbf{y}, \mathbf{z})$, where \mathbf{x} and \mathbf{y} are sequences of variables, B and H are conjunctions of atoms, with $\text{vars}(B) = \mathbf{x} \cup \mathbf{y}$, and $\text{vars}(H) = \mathbf{y} \cup \mathbf{z}$. B and H are respectively called the *body* and the *head* of R . A *contradiction-detecting dependency* (abbreviated CDD) is of the form $N : \forall \mathbf{x} B(\mathbf{x}) \rightarrow \perp$ where B is a conjunction of atoms, with $\text{vars}(B) = \mathbf{x}$. The body B may have equalities but no inequalities [12]. Inequalities are not used as they lead to undecidability even for TGDs [20]. Notice that whereas CDDs are a subset of DCs (Denial Constraints), they are different from Keys, FDs and EGDs (subsets of DCs).

A dependency with an empty body B and a non-empty head H is called a *fact*. Therefore, a fact is a set of atoms with existential variables (i.e. labeled nulls). A knowledge base $\mathcal{K} = (\mathcal{F}, \Sigma_T, \Sigma_C)$ consists of a finite sets of facts, TGDs and CDDs, respectively. Reasoning with a knowledge base is done via the chase. A rule $R : B \rightarrow H$ is *applicable* to a fact F if there exists a homomorphism π from B to F . The *application of R to F w.r.t. π* produces a finite set of atoms (also called atomset) $\alpha(F, R, \pi) = F \cup \pi(\text{safe}(H))$, where $\text{safe}(H)$ is obtained from H by replacing existential variables with fresh variables. The application of all TGDs to a set of facts is called the chase. The result of the chase on \mathcal{F} is denoted as $\mathcal{C}\ell_{\Sigma_T}(\mathcal{F})$ which produces an expanded set of facts \mathcal{F}^* . In this paper, we restrict ourselves to weakly-acyclic TGDs to avoid non-terminating chase sequences [15]. Let us consider the example in Figure 1 (b), on which we show the result of the chase.

Example 2.1. The result of the chase on the set of fact \mathcal{F}' is: $\mathcal{C}\ell_{\Sigma_T}(\mathcal{F}') = \mathcal{F}' \cup \{\text{prescribed}(\text{Nsaid}, \text{John})\}$.

Query Answering. Given a set of facts \mathcal{F} , an answer to Q in $\mathcal{K} = (\mathcal{F}, \Sigma_T, \Sigma_C)$ is a tuple of constants (A_1, \dots, A_k) such that there exists a homomorphism π from Q to $\mathcal{C}\ell_{\Sigma_T}(\mathcal{F})$, with $(A_1, \dots, A_k) = (\pi(x_1), \dots, \pi(x_k))$. We denote by $Q(\mathcal{F}, \Sigma_T)$ the set of all answers of Q over \mathcal{F} in presence of Σ_T .

Inconsistent knowledge bases and conflicts. A widely accepted assumption in KBs is that the set of TGDs is compatible with the set of CDDs, i.e. the union of the two sets is satisfiable [25]. A set of facts \mathcal{F} is inconsistent with respect to a set of TGDs Σ_T and CDDs Σ_C (or inconsistent for short) if and only if there exists a dependency $N \in \Sigma_C$ such that $\mathcal{C}\ell_{\Sigma_T}(\mathcal{F}) \models \text{body}(N)$. A knowledge base $\mathcal{K} = (\mathcal{F}, \Sigma_T, \Sigma_C)$ is inconsistent if and only if there exists a set of facts $\mathcal{F}' \subseteq \mathcal{F}$ such that \mathcal{F}' is inconsistent. We use the alternative notation $\mathcal{C}\ell_{\Sigma_T}(\mathcal{F}) \models \perp$ hereafter.

Example 2.2 (Example 2.1 Ct'd). The knowledge base $\mathcal{K} = (\mathcal{F}', \Sigma_T, \Sigma_C')$ is inconsistent because the bodies of the two CDDs are entailed from $\mathcal{C}\ell_{\Sigma_T}(\mathcal{F}')$.

Inconsistency can also be characterized by conflicts.

Definition 2.3 (Conflict). Let $\mathcal{K} = (\mathcal{F}, \Sigma_T, \Sigma_C)$ be an inconsistent knowledge base. A conflict is defined as a tuple $X=(N, h)$ such that h is a homomorphism from $\text{body}(N)$ to $\mathcal{C}\ell_{\Sigma_T}(\mathcal{F})$ such that $h(\text{body}(N)) \subseteq \mathcal{C}\ell_{\Sigma_T}(\mathcal{F})$.

Example 2.4. The knowledge base $\mathcal{K} = (\mathcal{F}', \Sigma_T, \Sigma_C')$ has two conflicts $X_1 = (N_1, h_1)$ and $X_2 = (N_2, h_2)$ defined as follows:

- $N_1 = \text{prescribed}(X, Y), \text{hasAllergy}(Y, X) \rightarrow \perp$.
- $h_1(X) = \text{Aspirin}, h_1(Y) = \text{John}$
- $N_2 = \text{prescribed}(X, Z), \text{prescribed}(Y, Z), \text{incompatible}(X, Y) \rightarrow \perp$.
- $h_2(X) = \text{Aspirin}, h_2(Y) = \text{Nsaid}$.

We denote by $\text{conflict}(\mathcal{K}, N)$ all the conflicts for a given constraint $N \in \Sigma_C$. The set of all conflicts of a given knowledge base is denoted as:

$$\text{allconflicts}(\mathcal{K}) = \bigcup_{N \in \Sigma_C} \text{conflict}(\mathcal{K}, N)$$

A knowledge base \mathcal{K} is consistent iff $\text{allconflicts}(\mathcal{K}) = \emptyset$.

In order for CDDs to be meaningful, we impose that CDDs contain atoms with join variables. This assumption is made to avoid for instance CDDs of the form $\text{prescribed}(X, Y) \rightarrow \perp$ in the above example. Such CDD is a schema constraint imposing that *prescribed* should be removed from the vocabulary of the KB.

3 UPDATE-BASED REPAIRING

In this section, we introduce the framework of update-based repairing for KBs. As opposed to deletion-based repairing, the granularity of update-based repairing is no longer an atom but instead a position that we need to update within a given atom. In what follows, we introduce the concept of a position and a fix on a position. Then, we proceed by giving the definition of a repair in such context, i.e. based on a minimal set of fixes needed to be applied in order to recover the consistency of a KB.

Given an atom $A = p(t_1, \dots, t_n)$, we denote by $\text{arity}(A) = n$ the arity of the predicate $\text{pred}(A) = p$. The tuple (A, i) such that $i \in [1, \text{arity}(A)]$ is called a position and *identifies* the position of the i -th argument of p . We denote by $\text{adom}(A, i, \mathcal{F})$ the active domain of the argument i of p in \mathcal{F} .

A position is a building block in update-based repairing as it gives access to the inner structure of an atom. For instance, $(A, 1)$ such that $A = \text{prescribed}(\text{Aspirin}, \text{John})$ is a position that refers to the first argument of A .

Given \mathcal{F} , the set of all positions of \mathcal{F} is defined as:

$$\text{pos}(\mathcal{F}) = \{(A, i) \mid A \in \mathcal{F} \text{ and } i \in [1, \text{arity}(A)]\}$$

The function $\text{value}_A^i(\mathcal{F})$ returns the value of the position (A, i) . Since existential variables can be present in atoms, $\text{value}_A^i(\mathcal{F})$ can be either an existentially quantified variable or a constant. The set of all values of a set of facts \mathcal{F} is defined as:

$$\text{vals}(\mathcal{F}) = \{\text{value}_A^i(\mathcal{F}) \mid (A, i) \in \text{pos}(\mathcal{F})\}.$$

A position fix specifies an update on a given atom in a given position.

Definition 3.1 (Position fix). A fix on a position (A, i) in \mathcal{F} is a triple (A, i, t) such that $t \in \text{adom}(A, i, \mathcal{F}) \setminus \text{value}_A^i(\mathcal{F})$ or $t = X_A^j$ is an existential variable that is uniquely attributed to (A, i) .

A fix on a position can specify a value that is within the active domain of the predicate p and different from the actual value. A fix can also specify an existential variable that refers to an unknown individual. Please note that such a variable is unique to the position in question and it is not used elsewhere in the knowledge base.

The *application* of a set of fixes \mathcal{P} on \mathcal{F} is defined as follows:

$$\text{apply}(\mathcal{F}, \mathcal{P}) = \{p(t'_1, \dots, t'_n) \mid A = p(t_1, \dots, t_n) \in \mathcal{F} \text{ and } \forall i \in \{1, \dots, n\} \text{ either } (A, i, t'_i) \in \mathcal{P} \text{ or } (A, i, t'_i) \notin \mathcal{P} \text{ and } t'_i = t_i\}$$

We consider only *valid* set of fixes which are set of fixes \mathcal{P} such that there exist no two fixes $(A, i, t), (A, i, t') \in \mathcal{P}$ and $t \neq t'$. The application of a set of fixes \mathcal{P} on a set of facts gives another set of facts called *the update* of \mathcal{F} by \mathcal{P} . It is clear that $|\mathcal{F}'| = |\mathcal{F}|$ and $\text{pos}(\mathcal{F}') = \text{pos}(\mathcal{F})$.

Example 3.2. The following is a set of fixes $\mathcal{P} = \{(A, 2, X_1), (A', 2, \text{Aspirin})\}$ such that:

- $A = \text{hasAllergy}(\text{John}, \text{Aspirin})$.
- $A' = \text{hasAllergy}(\text{Mike}, \text{Penicillin})$.

The update of \mathcal{F} by \mathcal{P} gives:

- $\mathcal{F}'_1 = \{\text{prescribed}(\text{Aspirin}, \text{John}), \text{hasAllergy}(\text{John}, X_1), \text{hasAllergy}(\text{Mike}, \text{Aspirin})\}$

The following set of fixes is not valid as it modifies the same position with different values:

- $\mathcal{P}' = \mathcal{P} \cup \{A, 2, \text{Penicillin}\}$

An important notion that will be used later is the reconstruction of a set of fixes \mathcal{P} given a set of facts \mathcal{F} and its update \mathcal{F}' . We define the function $\text{diff}(\mathcal{F}, \mathcal{F}')$ as follows:

$$\text{diff}(\mathcal{F}, \mathcal{F}') = \{(A, i, t'_i) \mid A = p(t_1, \dots, t_n) \in \mathcal{F}, A' = p(t'_1, \dots, t'_n) \in \mathcal{F}' \text{ and } \text{match}(A) = A' \text{ and } \exists j \in \{1, \dots, \text{arity}(A)\} \text{ such that } t'_j \neq t_j\}$$

Notice that the function $\text{match}(x)$ puts the atoms of \mathcal{F} and \mathcal{F}' in one-to-one correspondence. Such one-to-one correspondence exists because we know that \mathcal{F}' is an update of \mathcal{F} , therefore $|\mathcal{F}| = |\mathcal{F}'|$. $\text{match}(x)$ should satisfy the condition that $\text{match}(x) = y$ if and only if $x \in \mathcal{F}$ and $y \in \mathcal{F}'$ and $\text{pred}(x) = \text{pred}(y)$.

Example 3.3. Consider \mathcal{F} of Example 1.1 and its update \mathcal{F}' of Example 3.2, one can construct \mathcal{P} by defining $\text{match}(A_1) = A'_1$, $\text{match}(A_2) = A'_2$ and $\text{match}(A_3) = A'_3$ such that:

- $A_1 = \text{prescribed}(\text{Aspirin}, \text{John})$ and $A'_1 = \text{prescribed}(\text{Aspirin}, \text{John})$.
- $A_2 = \text{hasAllergy}(\text{John}, \text{Aspirin})$ and $A'_2 = \text{hasAllergy}(\text{John}, X_1)$.
- $A_3 = \text{hasAllergy}(\text{Mike}, \text{Penicillin})$ and $A'_3 = \text{hasAllergy}(\text{Mike}, \text{Aspirin})$.

Note that there may be finitely many one-to-one correspondences between two sets of facts.

The set of fixes \mathcal{P} gives a consistent update \mathcal{F}' . In fact, it is minimal in the sense that only what is necessary to recover consistency has been changed. In what follows, we introduce the notion of consistent fixes, repair fixes and update repair.

Definition 3.4 (c-fix and r-fix). Let \mathcal{K} be an inconsistent knowledge base, \mathcal{P} a set of fixes and $\mathcal{F}' = \text{apply}(\mathcal{F}, \mathcal{P})$ the update of \mathcal{F} by \mathcal{P} . \mathcal{P} is called consistent fixes (denoted c-fix) of \mathcal{K} iff $\mathcal{K} = (\mathcal{F}', \Sigma_T, \Sigma_C)$ is consistent. \mathcal{P} is called repair fixes (denoted r-fix) of \mathcal{K} iff \mathcal{P} is a c-fix and it contains no c-fix $\mathcal{P}' \subset \mathcal{P}$.

\mathcal{F}' is an update-repair if \mathcal{P} is an r-fix. A c-fix is a set of fixes that gives a consistent update, an r-fix is a set of fixes that gives a consistent update that is minimal with respect to the changes.

Example 3.5. \mathcal{P} is a c-fix and $\mathcal{P}_1 = \mathcal{P} \setminus \{(A', 2, \text{Aspirin})\}$ is an r-fix. However, $\mathcal{P}_2 = \mathcal{P} \setminus \{(A, 2, X_1)\}$ is not a c-fix.

The following is a u-repair produced by \mathcal{P}_1 :

$$\mathcal{F}'_1 = \{\text{prescribed}(\text{Aspirin}, \text{John}), \text{hasAllergy}(\text{John}, X_1), \text{hasAllergy}(\text{Mike}, \text{Penicillin})\}.$$

It is not hard to see that there exist finitely many r-fixes for a given set of facts \mathcal{F} because a position can take a finite set of values assuming that the active domain is finite.

After having defined the basic notions for update-based repairing, in what follows we introduce Π -repairability, a key concept in our framework. For a given knowledge base \mathcal{K} , we are interested in knowing whether there always exists a way to repair \mathcal{K} . In Example 1.1, the knowledge base is repairable because there exists an r-fix for \mathcal{F} . In fact, for an arbitrary inconsistent knowledge base \mathcal{K} , repairability is guaranteed as one can change all positions to fresh existential variables, and since such variables are unique to the positions no constraint will be triggered. This gives us a c-fix,

consequently an r-fix for \mathcal{K} . Π -repairability is a generalization of repairability where Π refers to those positions that are *immutable* or not allowed to be changed. This generalization helps us to know whether the KB is repairable when some positions are modified by the user and not allowed to be changed.

Definition 3.6 (Π -repairability). Let \mathcal{K} be an inconsistent knowledge base and $\Pi \subseteq \text{pos}(\mathcal{F})$ be a set of positions. We say that \mathcal{K} is Π -repairable if and only if there exists an r-fix \mathcal{P} of \mathcal{K} such that there exists no $(A, i, t) \in \mathcal{P}$ and $(A, i) \in \Pi$.

A knowledge base can be inconsistent but Π -repairable. In such case, Π -repairability indicates in a sense the possibility of finding a u-repair for \mathcal{K} if certain positions are fixed prior to the repairing process. If \mathcal{K} is not Π -repairable then \mathcal{K} has no u-repair whose corresponding r-fix \mathcal{P} changes the positions in $\text{pos}(\mathcal{F}) \setminus \Pi$.

As stated above, Π -repairability is a generalization of the concept of repairability. When all positions are immutable then Π -repairability reduces down to a consistency check. Formally, if $\Pi = \text{pos}(\mathcal{F})$ and \mathcal{K} is Π -repairable then \mathcal{K} is consistent.

Algorithm 1 for checking Π -repairability proceeds by changing all positions to fresh existential variables except those positions that belong to Π . Then, we check the consistency of this new knowledge base using $\text{CHECKCONSISTENCY}(\mathcal{K})$. In fact, the algorithm checks if fixing some positions with their corresponding values will result in fixing the violations of some CDDs. If this is the case, the knowledge base can never be repaired.

Example 3.7. Consider the following knowledge base \mathcal{K} with an empty Σ_T :

- $\mathcal{F} = \{p(a, b), q(b, d)\}$
- $\Sigma_C = \{p(X, Y), q(Y, Z) \rightarrow \perp\}$

If we take $\Pi = \emptyset$ then \mathcal{K} is Π -repairable. This is because the c-fix $\mathcal{P} = \{(p(a, b), 1, X_1), (p(a, b), 2, X_2), (q(b, d), 1, X_3), (q(b, d), 1, X_4)\}$ gives a consistent update. Consequently, \mathcal{P} is a c-fix. Necessarily, one can consider the r-fix $\mathcal{P}' = \{(p(a, b), 2, X_1)\} \subset \mathcal{P}$ which gives a u-repair. However, if $\Pi = \{(p(a, b), 2), (q(b, d), 1)\}$ then \mathcal{K} is not Π -repairable because regardless of the values that the other positions can take the dependency will always be violated. Note that the fact that Σ_T is empty does not change the situation, given that the consistency check function is generic.

Checking Π -repairability is easy from a computational perspective. Algorithm 1 does perform such check in a polynomial time. The function $\text{CHECKCONSISTENCY}(\mathcal{K})$ in Algorithm 1 evaluates on the body of every CDD $N \in \Sigma_C$ on $\text{Cl}_{\Sigma_T}(\mathcal{F})$ and checks whether the query has an answer. If this is the case, \mathcal{K} is inconsistent, otherwise it proceeds until no CDD is left to be evaluated, where the knowledge base achieves consistency. Clearly, the function $\Pi\text{-REP}(\mathcal{K})$ runs in linear time of the size of $\text{pos}(\mathcal{F})$ plus the computational overload of the function $\text{CHECKCONSISTENCY}(\mathcal{K})$. This gives a polynomial data complexity as evaluating boolean conjunctive queries is polynomial in data complexity even in presence of weakly-acyclic TGDs [12, 22].

We now need to prove that the algorithm is sound, i.e. if the knowledge base is Π -repairable then the algorithm produces true as an output, otherwise false.

PROPOSITION 3.8. \mathcal{K} is Π -repairable iff $\Pi\text{-rep}(\mathcal{K}, \Pi) = \text{true}$.

PROOF. (\Rightarrow): suppose that \mathcal{K} is Π -repairable and $\Pi\text{-REP}(\mathcal{K}, \Pi)$ returns false. The former implies that there exists an r-fix \mathcal{P}' of \mathcal{K} such that $\mathcal{F}'' = \text{apply}(\mathcal{F}, \mathcal{P}')$ is the u-repair of \mathcal{F} By \mathcal{P}' . The latter implies that $\mathcal{K}' = (\mathcal{F}', \Sigma_T, \Sigma_C)$ in line 5 is inconsistent, thus there exists a conflict $\mathcal{X} = (N, h)$ in \mathcal{K}' . Since there

exists a homomorphism from $\text{body}(N)$ to \mathcal{F}' , we now show that $\mathcal{K}'' = (\mathcal{F}'', \Sigma_T, \Sigma_C)$ is necessarily inconsistent by constructing a homomorphism g from \mathcal{F}' to \mathcal{F}'' i.e. \mathcal{X} would also be a conflict in \mathcal{K}'' , thus \mathcal{K}'' is inconsistent.

Recall that \mathcal{P}' is the set of fixes that assigns to every position $(A, i) \in \text{pos}(\mathcal{F})$, $(A, i) \notin \Pi$ a unique existential variable X_A^i . Let $\mathcal{P}'' = \text{diff}(\mathcal{F}', \mathcal{F}'')$, we define the homomorphism $g : A \mapsto B$ such that $A = \{X_A^i \mid (A, i, X_A^i) \in \mathcal{P}'\}$, $B = \{t \mid (A, i, t) \in \mathcal{P}''\}$, and $g(X_A^i) = t$ such that $(A, i, t) \in \mathcal{P}''$. Since there exists a homomorphism from $\text{body}(N)$ to \mathcal{F}' , and from \mathcal{F}' to \mathcal{F}'' then there exists necessarily a homomorphism from $\text{body}(N)$ to \mathcal{F}'' . Hence, \mathcal{K}'' is inconsistent.

(\Leftarrow): it is trivial, if \mathcal{K}' is consistent then \mathcal{P} is a c-fix of \mathcal{K} such that $\nexists(A, i, t) \in \mathcal{P}$, (A, i) . By definition, $\exists \mathcal{P}' \subseteq \mathcal{P}$ such that \mathcal{P}' is an r-fix of \mathcal{K} . \square

Algorithm 1 Π -repairability

```

1: function  $\Pi\text{-REP}(\mathcal{K}, \Pi)$ 
2:    $\Pi' \leftarrow \text{pos}(\mathcal{F}) \setminus \Pi$ 
3:    $\mathcal{P} \leftarrow \{(A, i, t) \mid (A, i) \in \Pi' \text{ and } t = X_A^i\}$ 
4:    $\mathcal{F}' \leftarrow \text{apply}(\mathcal{F}, \mathcal{P})$ 
5:    $\mathcal{K}' \leftarrow (\mathcal{F}', \Sigma_T, \Sigma_C)$ 
   return  $\text{CHECKCONSISTENCY}(\mathcal{K}')$ 
6: end function

```

We have introduced so far the key concepts of our framework. Nevertheless, as already mentioned in the introduction, update-based repairing is unfeasible in practice because there are no guidelines on (1) how to choose the positions among those possible, and (2) who provides the corresponding fixes. Our positioning here is that update-based repairing should go hand in hand with user intervention. In the next section, we introduce our interactive framework serving this purpose.

4 USER INTERVENTION

The key idea behind user intervention is that the user may have a repair in mind, which corresponds to how the knowledge base should turn to be consistent. Obviously, it is impossible for a user to manually repair the KB. In this section, we propose a framework of *inquiry dialogue* that takes a place between the knowledge base and the user. The basic idea is that the knowledge base asks questions about some fixes and the user chooses which one is true until he reaches a consistent knowledge base or, alternatively, a u-repair under some conditions.

Definition 4.1 (*Inquiry*). Given an inconsistent knowledge base \mathcal{K} and a possibly empty set of positions Π . A **question** has the form $\phi = \{f_1, \dots, f_n\}$ such that f_k is a fix. An **answer** to ϕ is a fix $f_k \in \phi$. Given a conflict $\mathcal{X} = (N, h)$ in \mathcal{K} , a question $\phi = \{f_1, \dots, f_n\}$ is said to be **sound** if and only if for every fix $f_k = (A, i, t) \in \phi$ where $\Pi' = \Pi \cup \{(A, i)\}$, $\mathcal{K} = (\text{apply}(\mathcal{F}, \{f\}), \Sigma_T, \Sigma_C)$ is Π' -repairable. An **inquiry** over \mathcal{K} is a finite sequence of pair of sound questions and answers $\Omega_{\mathcal{K}} = ((\phi_1, f_1), \dots, (\phi_n, f_n))$ such that $f_i \in \phi_i$.

A question ϕ is a set of fixes, whereas an answer is a fix that the user chooses from ϕ . In the framework, questions are sound if, once answered, will not render the knowledge base unrepairable. These questions are crucial to guide the user.³

Example 4.2. Consider the knowledge base of Example 1.1 and the following sound question:

³Hereafter, every question is meant to be sound.

- $\phi = \{(A, 1, X_1), (A, 2, X_2), (A', 1, \text{Mike}), (A', 1, X_3), (A', 2, \text{Penicillin}), (A', 2, X_4)\}$ such that:
 - $A = \text{prescribed}(\text{Aspirin}, \text{John})$ and,
 - $A' = \text{hasAllergy}(\text{John}, \text{Aspirin})$.

An inquiry is a sequence of tuples of question and answer. In what follows we show how a sound question can be generated and how an inquiry with a user takes place.

Algorithm 2 generates a sound question from a given conflict \mathcal{X} . The choice of a conflict being the starting point of a question is evident. In fact, fixing those atoms that are parts of some conflicts necessarily solves inconsistencies. The algorithm in line 4 generates all positions of the atoms of the conflict \mathcal{X} , then for each position (A, i) that does not belong to Π , we generate all possible fixes in lines 6-7. The fixes change the value of the position (A, i) to other values in the active domain different than the actual value and to an existential variable uniquely attributed to (A, i) . Next in line 10, we enter in a filtering step where each fix is omitted if it renders the knowledge base not Π -repairable. Then it returns just ϕ . The following lemma proves that Algorithm 2 always gives a non-empty question which is necessarily sound.

LEMMA 4.3. *Given an inconsistent knowledge base \mathcal{K} and a set of positions Π such that \mathcal{K} is Π -repairable. Given a conflict $\mathcal{X} = (N, h)$, then $\text{soundquestion}(\mathcal{K}, \Pi, \mathcal{X}) \neq \emptyset$ and $\text{soundquestion}(\mathcal{K}, \Pi, \mathcal{X})$ outputs a sound question.*

PROOF. First, if $\Pi = \text{pos}(\mathcal{F})$ then \mathcal{K} is consistent (Π -repairability reduces down to consistency in this case), therefore there will be no conflict \mathcal{X} in \mathcal{K} . Assume that $\Pi \subset \text{pos}(\mathcal{F})$, then $\text{SOUNDQUESTION}(\mathcal{K}, \Pi, \mathcal{X}) = \emptyset$ if and only if: (1) In Line 5, $\Pi' \subseteq \Pi$, or, (2) In Line 8, $val = \emptyset$ for each position $(A, i) \in P$, or, (3) In Line 16, every fix is removed from ϕ .

For (1), suppose it is the case. We know that \mathcal{K} is Π -repairable, therefore there exists an r-fix \mathcal{P} of \mathcal{K} such that there exists no $(A, i, t) \in \mathcal{P}$ and $(A, i) \in \Pi$. Let $\mathcal{F}' = \text{apply}(\mathcal{F}, \mathcal{P})$ be the update repair of \mathcal{F} by \mathcal{P} . We know that $h(\text{body}(N)) \subseteq \mathcal{F}$, and $\forall A \in h(\text{body}(N))$ and for every $j \in [1, \text{arity}(A)]$, $(A, i) \notin \mathcal{P}$ for some t because $(A, i) \in \Pi'$. Therefore, $h(\text{body}(N)) \subseteq \mathcal{F}'$, which means that $\mathcal{X} = (N, h)$ is a conflict in $\mathcal{K} = (\mathcal{F}', \Sigma_T, \Sigma_C)$, consequently \mathcal{K}' is inconsistent and \mathcal{F}' is not a u-repair, thus \mathcal{P} is not r-fix, which contradicts the fact that \mathcal{K} is Π -repairable. For (2), it cannot be the case because val can always hold $\{X_A^i\}$. For (3), it is clear at each iteration the fix $f_k = (A, i, X_A^i)$ is in ϕ and will not be removed because if \mathcal{K}' is Π -repairable then $\mathcal{K}' = (\text{apply}(\mathcal{F}, f_k), \Sigma_T, \Sigma_C)$ is also Π' -repairable where $\Pi' = \Pi \cup \{(A, i)\}$.

The fact that $\text{SOUNDQUESTION}(\mathcal{K}, \Pi, \mathcal{X})$ returns a sound question is quite straightforward since if $\text{SOUNDQUESTION}(\mathcal{K}, \Pi, \mathcal{X}) \neq \emptyset$, the Algorithm in line 14 drops any answer that does not lead to a Π -repairable knowledge base. \square

This lemma tells us that if \mathcal{K} is Π -repairable, we can always find a sound question. This relies on the intuition that \mathcal{K} is Π -repairable, i.e. there necessarily exists some fixes that can be applied to render the knowledge base consistent.

Engaging the user in an inquiry needs to guarantee that the knowledge base is eventually repaired the way the user want it to be. However, such goal may never be accomplished if the inquiry cannot ensure that the resulting knowledge base is consistent. Algorithm 3 is the principled procedure that undertakes an inquiry dialogue with the user. The key idea is that we keep asking questions until there is no conflict left in the knowledge base. When

the user chooses a fix (A, i, t) from ϕ , the position (A, i) becomes *immutable* to prevent modifying the value again. The algorithm terminates and produces a consistent knowledge base.

Algorithm 2 Generate sound question

```

1: function SOUNDQUESTION( $\mathcal{K}, \Pi, \mathcal{X}$ )
2:    $\mathcal{X} = (N, h)$ 
3:    $\phi \leftarrow \emptyset$ 
4:    $\Pi' \leftarrow \{(A, i) \mid A \in h(\text{body}(N)) \text{ and } i \in [1, \text{arity}(A)]\}$ 
5:   for each  $(A, j) \in \Pi' \setminus \Pi$  do
6:      $val \leftarrow \text{adom}(A, i, \mathcal{F}) \setminus \{\text{value}_A^i(\mathcal{F})\}$ 
7:      $val \leftarrow val \cup \{X_A^i\}$ 
8:      $\phi \leftarrow \phi \cup \{(A, i, t) \mid t \in val\}$ 
9:   end for
10:  for each  $f_k = (A, i, t) \in \phi$  do
11:     $\Pi_{tmp} \leftarrow \Pi \cup \{(A, i)\}$ 
12:     $\mathcal{K}' \leftarrow (\text{apply}(\mathcal{F}, f), \Sigma_T, \Sigma_C)$ 
13:    if  $\Pi\text{-REP}(\mathcal{K}', \Pi_{tmp}) = \text{false}$  then
14:       $\phi \leftarrow \phi \setminus (A, i, t)$ 
15:    end if
16:  end for
17:  return  $\phi$ 

```

Algorithm 3 Inquiry with a user

```

1: function INQUIRY( $\mathcal{K}, \Pi$ )
2:    $\mathcal{K}' \leftarrow \mathcal{K}$ 
3:    $\Pi' \leftarrow \Pi$ 
4:   while  $\text{allconflicts}(\mathcal{K}') \neq \emptyset$  do
5:     Pick a conflict  $\mathcal{X} \in \text{allconflicts}(\mathcal{K}')$ 
6:      $\phi \leftarrow \text{SOUNDQUESTION}(\mathcal{K}, \Pi', \mathcal{X})$ 
7:      $f \leftarrow \text{ASKUSER}(\phi)$ 
8:      $\mathcal{K}' \leftarrow (\text{apply}(\mathcal{F}, f), \Sigma_T, \Sigma_C)$ 
9:      $\Pi' \leftarrow \Pi' \cup \{(A, i) \mid f = (A, i, t)\}$ 
10:    Recompute  $\text{allconflicts}(\mathcal{K}')$ 
11:  end while
12:  return  $\mathcal{K}'$ 

```

PROPOSITION 4.4 (SOUNDNESS AND TERMINATION). *Given an inconsistent knowledge base \mathcal{K} and a set of positions Π . Then, $\text{inquiry}(\mathcal{K}, \Pi)$ returns a consistent KB \mathcal{K} in a finite time.*

PROOF. Let $\mathcal{K}'_i, \Pi'_i, \phi_i, f_i$ be the knowledge base \mathcal{K}' , the set of positions Π' , the sound question ϕ and the chosen fix f at the beginning of the while loop at round i .

Round 1: $\mathcal{K}'_1 = \mathcal{K}$ is inconsistent and Π'_1 -repairable such that $\Pi'_1 = \Pi$.

Round i : \mathcal{K}'_i is either consistent, therefore $\text{allconflicts}(\mathcal{K}') = \emptyset$ and Algorithm 3 terminates, or inconsistent. However, we know that it is Π_i -repairable because $\mathcal{F}'_i = \text{apply}(\mathcal{F}'_{i-1}, \{f_i\})$ such that $f_i \in \phi_i$ and ϕ_i is a sound question. Let this round be the one in which $|\text{pos}(\mathcal{F}'_i)| - |\Pi'_i| = 1$. This means that at line 6 $\text{SOUNDQUESTION}(\mathcal{K}, \Pi, \mathcal{X}) \neq \emptyset$ and when the user chooses a fix in line 7, it is clear that at line 11 \mathcal{K}'_i is Π'_i -repairable with $\Pi'_i = \text{pos}(\mathcal{F}'_i)$. It is obvious that \mathcal{K}'_i is consistent hence $\text{allconflicts}(\mathcal{K}') = \emptyset$. \square

In what follows, we investigate the complexity of Algorithm 2.

PROPOSITION 4.5. *In the worst-case, Algorithm 2 runs in $O(d \times (|\text{pos}(\mathcal{F})| + C_{\Pi\text{-rep}}))$ with d being the size of the largest active domain in \mathcal{K} and $C_{\Pi\text{-rep}}$ being the worst-case complexity of Π -repairability algorithm.*

PROOF. The worst-case corresponds to $\Pi = \emptyset$ and $h(\text{body}(N)) = \mathcal{F}$, i.e. the whole set of facts is a conflict. In this case, the loop at line 5 will iterate over all positions, i.e. $\text{pos}(\mathcal{F})$. The loop at line 8 depends on d . The additional loop at line 10 performs d iterations.

We assume that the instruction at line 12 runs in constant time. Then, the function $\Pi\text{-REP}(\mathcal{K}', \Pi'_{tmp})$ is called d times. \square

The ultimate and most desirable goal of the inquiry is to arrive at the user's repair. A well-founded framework is the one that meets such requirement. However, this depends on how the user answers the questions, his background knowledge and so on. Therefore, some assumptions have to be made. In the next section, we consider a special case, i.e. when the user is *an oracle*.

4.1 The Oracle

In this section, we discuss the case in which interaction takes place with an oracle O . The oracle corresponds to a u-repair \mathcal{F}_O with an associated answering mechanism. The oracle draws its answers from \mathcal{F}_O as follows: given a question ϕ , f_i is an oracle answer if and only if $f_i \in \text{diff}(\mathcal{F}, \mathcal{F}_O)$. In case of multiple answers from the oracle, O non-deterministically chooses one of them. Note that not all the sets of fixes in $\text{diff}(\mathcal{F}, \mathcal{F}_O)$ are necessarily r-fixes. There may exist finitely many set of fixes, even though we assume that if a given r-fix \mathcal{P}_O is chosen by O , we name it an oracle r-fix. Note that such r-fix always exists as shown hereafter.

PROPOSITION 4.6. *Let \mathcal{F}' be a u-repair of \mathcal{F} . Then, there exists a one-to-one correspondence $\text{match}(x)$ such that $\text{diff}(\mathcal{F}, \mathcal{F}')$ is an r-fix.*

The proof is straightforward as there may exist exponentially many $\text{match}(x)$ that make all possible one-to-one correspondences. One of them must necessarily correspond to the real match because $|\mathcal{F}| = |\mathcal{F}'|$ and \mathcal{F}' is homomorphic to \mathcal{F} .

It turns out that when interacting with the oracle, the oracle is capable of answering every question asked by Algorithm 3.

LEMMA 4.7. *Given a consistent knowledge base \mathcal{K} , a possibly empty set of positions Π , an oracle O and its chosen r-fix \mathcal{P}_O . Every question ϕ_i generated in $\text{inquiry}(\mathcal{K}, \Pi)$ contains at least a fix f_i such that f_i is in \mathcal{P}_O .*

PROOF. If there exists a sound question ϕ_i generated by $\text{INQUIRY}(\mathcal{K}, \Pi)$ at an iteration i such that $\phi_i \cap \mathcal{P}_O = \emptyset$ then there exists $f \in \mathcal{P}_O$ such that $\mathcal{K} = (\text{apply}(\mathcal{F}, \{f\}), \Sigma_T, \Sigma_C)$ is not Π'_i -repairable. Therefore, \mathcal{K} has no u-repair. This contradicts the fact that \mathcal{F}_O is a u-repair. \square

This lemma gives us the most important result, by stating that when the inquiry ends, the resulting knowledge base is in fact the oracle's u-repair \mathcal{F}_O .

PROPOSITION 4.8 (SOUNDNESS W.R.T O). *Let $\mathcal{K}' = (\mathcal{F}', \Sigma_T, \Sigma_C)$ be the knowledge base returned by $\text{inquiry}(\mathcal{K}, \Pi)$ with an oracle O as the user. Then, \mathcal{F}' is the oracle's repair \mathcal{F}_O .*

PROOF. Since every question ϕ_i contains at least a fix $f \in \mathcal{P}_O$ then O will definitely choose a fix $f \in \mathcal{P}_O$. After answering by f , every next question ϕ_{i+1} will not contain f because once a fix is applied it will never be proposed again. However, by Lemma 4.7, ϕ_{i+1} will definitely contain a fix f' such that $f' \in \mathcal{P}_O \setminus \{f\}$. Hence, O will choose it until choosing all fixes in \mathcal{P}_O . We can see that in fact we are applying \mathcal{P}_O on \mathcal{F} one fix at a time. We know that \mathcal{P}_O is an r-fix, thus in other words we are constructing a u-repair identical to \mathcal{F}_O . Therefore, $\mathcal{F}' = \mathcal{F}_O$. \square

Let us give an example of an inquiry with an oracle.

Example 4.9 (Inquiry with oracle). Consider the knowledge base of Figure 1 (b) and the oracle repair \mathcal{F}_O :

$$\mathcal{F}_O = \begin{cases} \text{prescribed}(\text{Aspirin}, \text{John}) & \text{hasAllergy}(\text{Mike}, \text{Aspirin}) \\ \text{hasAllergy}(\text{Mike}, \text{Penicillin}) & \text{hasPain}(\text{Mike}, \text{Migraine}) \\ \text{isPainKillerFor}(\text{Nsaid}, \text{Migraine}) \\ \text{incompatible}(\text{Aspirin}, \text{Nsaid}) \end{cases}$$

The inquiry is as follows:

- (1) KB: which fix is true from the following set?
 - $\{\text{prescribed}(\text{Aspirin}, \text{John}), 1, t \mid t \in \{X_1, \text{Nsaid}\}\} \cup$
 - $\{\text{prescribed}(\text{Aspirin}, \text{John}), 2, t \mid t \in \{X_2, \text{Mike}\}\} \cup$
 - $\{\text{hasAllergy}(\text{John}, \text{Aspirin}), 1, t \mid t \in \{X_3, \text{Mike}\}\} \cup$
 - $\{\text{hasAllergy}(\text{John}, \text{Aspirin}), 2, t \mid t \in \{X_4, \text{Penicillin}\}\}$
- (2) O : the fix $(\text{hasAllergy}(\text{John}, \text{Aspirin}), 1, \text{Mike})$ is true.
- (3) KB: which fix is true from the following set?
 - $\{\text{incompatible}(\text{Aspirin}, \text{Nsaid}), 1, X_5\} \cup$
 - $\{\text{incompatible}(\text{Aspirin}, \text{Nsaid}), 2, X_6\} \cup$
 - $\{\text{prescribed}(\text{Aspirin}, \text{John}), 1, t \mid t \in \{X_7, \text{Nsaid}\}\} \cup$
 - $\{\text{prescribed}(\text{Aspirin}, \text{John}), 2, X_8\} \cup$
 - $\{\text{hasPain}(\text{John}, \text{Migraine}), 1, X_9\} \cup$
 - $\{\text{hasPain}(\text{John}, \text{Migraine}), 2, X_{10}\} \cup$
 - $\{\text{isPainKillerFor}(\text{Nsaid}, \text{Migraine}), 1, X_{11}\} \cup$
 - $\{\text{isPainKillerFor}(\text{Nsaid}, \text{Migraine}), 2, X_{12}\}$
- (4) O : the fix $(\text{hasPain}(\text{John}, \text{Migraine}), 1, \text{Mike})$ is true.

The knowledge base asks a question on a possible set of fixes. Then, the oracle chooses among them a fix that belongs to its r-fix. As one can notice after applying the fixes provided by the oracle in 2 and 4, the resulting knowledge base is indeed consistent and its set of facts equals \mathcal{F}_O . Notice that for instance the fix $f = (\text{incompatible}(\text{Aspirin}, \text{Nsaid}), 1, X_5)$ has no proposed value other than the existential variable because the active domain is empty. An additional comment is in order. The size of the questions grows polynomially (and not exponentially) in the size of the conflicts and in the size of the active domain. As a consequence, presenting these questions to the user is an implementation concern that can benefit from advanced HCI techniques [8] and is beyond the scope of this paper.

When interacting with an oracle, Algorithm 1 performs as many iterations as the number of conflicts in the knowledge base. However, the number of iterations corresponds to the size of the oracle's r-fix, denoted as r_O^{num} . This is the case because the oracle at each step answers with a fix $f \in \mathcal{P}_O$ until all fixes in \mathcal{P}_O are used. Given a set of empty positions Π and a Π -repairable inconsistent knowledge base $\mathcal{K} = (\mathcal{F}, \Sigma_T, \Sigma_C)$, then $r_O^{\text{num}} \leq |\text{pos}(\mathcal{F})|$. Therefore, the number of iterations is linear in the size of \mathcal{F} . However, the algorithm is dominated by the complexity of $\text{SOUNDQUESTION}(\mathcal{K}, \Pi', X)$ at line 6 and the computation of all conflicts. In fact, a conflict is the result of evaluating the body of a CDD, thus leading to a polynomial data complexity of boolean conjunctive query (for weakly-acyclic TGDs).

PROPOSITION 4.10. *Let C_{query} be the data complexity of evaluating a conjunctive query on a set of facts \mathcal{F} , in presence of a set of weakly-acyclic TGDs Σ_T . C_{soundq} be the complexity of soundquestion (Proposition 4.5) then the complexity of Algorithm 1 is $O(|\text{pos}(\mathcal{F})| \times (|\Sigma_C| \times C_{\text{query}} + C_{\text{soundq}}))$.*

The above result gives us a polynomial delay algorithm.

COROLLARY 4.11. *Algorithm 3 takes a polynomial delay between questions.*

A polynomial delay algorithm is an algorithm in which the time between the output of the solutions is bounded by a polynomial function of the input size in the worst case [23]. In between questions, we perform query evaluation which costs $|\Sigma_C| \times C_{\text{query}}$ and the computation of a sound question which costs C_{soundq} . Therefore, the user will not have to wait from one question to the next more than an amount of time that is polynomially bounded.

5 QUESTIONING STRATEGIES

The goal of a strategy is to minimize the number of questions to be asked to the user in order to arrive at a consistent knowledge base. In this section, we present four strategies improving one on another. We introduce: the baseline strategy called *random*; another strategy, called *opti-join*, that improves over *random* by considering the so-called join positions; another variant of *opti-join* called *opti-prop* that uses propagation, and finally a fourth strategy, called *opti-mcd* that improves over *opti-join*.

First, let us define the lower and upper bounds of the number of questions for each strategy. It is obvious that the maximum number of questions is equal to $|\text{pos}(\mathcal{F})|$. This case corresponds to a knowledge base in which every position needs to be changed to recover consistency. The minimum number of questions is clearly zero if the knowledge base is consistent.

The functions $\Pi\text{-REP}(\mathcal{K}, \Pi)$ of Π -repairability and $\text{CHECKCONSISTENCY}(\mathcal{K}')$ in Algorithm 1 and $\text{recompute_allconflicts}(\mathcal{K}')$ in Algorithm 3 shown in Section 4 are used in all the strategies. In the following, we detail Algorithm 4 where we propose an optimized version of these functions.

CHECKCONSISTENCY-OPT(\mathcal{K}'): the most naive approach for consistency check is to compute the chase on \mathcal{F} to get $\mathcal{C}\ell_{\Sigma_T}(\mathcal{F})$ then to check whether there exists a CDD whose body evaluates to true in $\mathcal{C}\ell_{\Sigma_T}(\mathcal{F})$, as implemented in $\text{CHECKCONSISTENCY}(\mathcal{K}')$. The optimized version $\text{CHECKCONSISTENCY-OPT}(\mathcal{K}')$ considers CDDs and TGDs such that \perp is seen as unary predicate (i.e. a constant). If, during the chase, the constant \perp is produced then the knowledge base is inconsistent. This is quite fruitful as it helps to stop consistency check as early as possible.

$\Pi\text{-REPOPT}(\mathcal{K}, \Pi)$: we can easily observe that if a knowledge base \mathcal{K} is Π -repairable and some positions have been fixed using a set of values \mathcal{V} then in the case in which a new fix f arrives with value v (constant or fresh existential variable), the knowledge base stays Π -repairable if $v \notin \mathcal{V}$. This is quite intuitive because if the fixed positions do not trigger any CDDs, the new value will not trigger any CDDs since all atoms have different values. If the value is already used, we proceed to the optimized consistency check $\text{CHECKCONSISTENCY-OPT}(\mathcal{K}')$. This is the optimized Π -repairability check of $\Pi\text{-REP}(\mathcal{K}, \Pi)$.

Let us now turn to the optimization of $\text{allconflicts}(\mathcal{K}')$. Let $\text{allconflicts}_{\text{naive}}(\mathcal{K}')$ be defined as the set of all naive conflicts. A naive conflict $X = (N, h)$ is defined as a conflict in the sense of Definition 2.3 except that h is a homomorphism from $\text{body}(N)$ to \mathcal{F} such that $h(\text{body}(N)) \subseteq \mathcal{F}$ and $\mathcal{C}\ell_{\Sigma_T}(\mathcal{F}) \models \perp$. These conflicts are computed on \mathcal{F} without applying the chase. It is clear that if $\text{allconflicts}_{\text{naive}}(\mathcal{K}') = \emptyset$, \mathcal{K} is not necessarily consistent as there is the possibility of having conflicts that will appear after applying the chase like the conflict X_2 in Example 2.4. However, we observed that resolving naive conflicts at first can eliminate other conflicts that are discovered after applying the chase. For instance in Example 2.4, if we resolve the conflict X_2 by updating the atom $\text{prescribed}(\text{Aspirin}, \text{John})$ on the first position, this will resolve the conflict that can be detected using the second CDD after applying the TGD. Therefore, our strategies are *two-phases* strategies. In the first phase, naive conflicts are resolved, while in the second phase, if the KB is still inconsistent, new conflicts are discovered and resolved during the chase. In what follows, we provide an optimization of conflicts computation.

UPDATECONFLICTS(\mathcal{K}'): we compute the initial set of naive conflicts over \mathcal{K} and keep them in a set C_{naive} . Then C_{naive} is

updated as follows. If the user provides a fix $f = (A, i, t)$ which results in a new set of facts \mathcal{F}' , then we remove all conflicts that are related to A from C_{naive} . Next, we define a subset $\Sigma_C^A \subseteq \Sigma_C$ that is related to A as follows: a CDD $N \in \Sigma_C^A$ iff $\exists A' \in \text{body}(N)$ and a homomorphism h such that $h(A') = A$. Finally, C_{naive} is updated by evaluating the body of each CDD $N \in \Sigma_C^A$ over the new set of facts \mathcal{F}' . In this optimization, instead of recomputing all conflicts, we are limiting the computation to the modified atom which is more efficient than evaluating all CDDs on \mathcal{F}' .

Once we have defined the above optimizations, we turn our attention to our proposed strategies, namely *random*, *opti-join*, *opti-prop* and *opti-mcd*. Algorithms 4 & 5 are parametrized, thus we can easily plug in the above optimizations. The main code of each strategy is Algorithm 4 which calls the functions $\text{GENERATEQUESTION}(\mathcal{K}, \Pi', \mathcal{X})$ and $\text{GENERATEQUESTION-CHASE}(\mathcal{K}, \Pi', \mathcal{X})$. These functions are implemented differently for each strategy. In addition, these functions make use of $\text{SOUNDQUESTION}(\mathcal{K}, \Pi', \mathcal{X})$ for which the function $\text{RETRIEVE-POSITIONS}(\mathcal{X}, \mathcal{K})$ changes from a strategy to another. For space reasons, in the following we report a concise description of each strategy. Their implementation and effectiveness are discussed in the next section.

Random. This strategy selects randomly a conflict from C_{naive} before asking a question about all positions. More precisely, $\text{GENERATEQUESTION}(\mathcal{K}, \Pi', \mathcal{X})$ randomly picks a conflict from C_{naive} and calls $\text{SOUNDQUESTION}(\mathcal{K}, \Pi', \mathcal{X})$. Then, $\text{RETRIEVE-POSITIONS}(\mathcal{X}, \mathcal{K})$ for each atom in $h(\text{body}(N))$, generates all positions (A, i) and proceed normally in $\text{SOUNDQUESTION}(\mathcal{K}, \Pi', \mathcal{X})$. While applying the chase if a violation of a CDD is detected, $\text{GENERATEQUESTION-CHASE}(\mathcal{K}, \Pi', \mathcal{X})$ gets all facts in \mathcal{F} that contribute to its violation, then generates all positions from this set and returns it as a question using $\text{SOUNDQUESTION}(\mathcal{K}, \Pi', \mathcal{X})$.

Opti-join. This strategy improves over *random* on $\text{RETRIEVE-POSITIONS}(\mathcal{X}, \mathcal{K})$ where only join positions are generated. Given a conflict $X = (N, h)$, a position (A, i) is a join position if and only if the variable at the position i in A' is a join variable in the CDD N such that $h(A') = A$. For instance, consider the example of Figure 1(b) the position $(\text{prescribed}(\text{Aspirin}, \text{John}), 1)$ is a join position because the variable X in $A' = \text{prescribed}(X, Y)$ is a join variable in the second CDD. Clearly, this strategy generates smaller questions and most notably avoid asking unnecessary questions. Consider the knowledge base \mathcal{K} without TGDs:
 $\mathcal{F} = \{\text{isUrgent}(\text{Mike}, a, 145), \text{isDeferredTo}(\text{Mike}, 12/10/2015)\}$.
 $\Sigma_C = \{\text{isUrgent}(X, Y, Z), \text{isDeferredTo}(X, W) \rightarrow \perp\}$.

Here the position $(\text{isDeferredTo}(\text{Mike}, "12/10/2015"), 2)$ is not a join position. However, the position $(\text{isUrgent}(\text{Mike}, a, 145), 1)$ is a join position. Join positions are pivotal in order to resolve conflicts, since changing non-join positions does not affect the homomorphisms and does not resolve conflicts.

Opti-prop. This strategy behaves the same as *opti-join* except that a propagation technique is used. By definition if the user chooses a fix $f = (\text{isUrgent}(\text{Mike}, a, 145), 1, X_1)$ from a question ϕ produced from a conflict X , then every position generated from X (except the chosen one in f) is added to Π if and only if it is not involved in any other conflict X' . This is quite intuitive because when the user chooses a fix f from a question ϕ , he is implicitly indicating that they are non-erroneous. However, if some of these positions participate in other conflicts then it is possible that they are erroneous, thus they will not be added to Π .

Opti-mcd. This strategy is an improvement over *opti-join*, it is based on the so-called Conflict Hypergraph (CH) [13, 24] where $\text{GENERATEQUESTION}(\mathcal{K}, \Pi', \mathcal{X})$ and $\text{GENERATEQUESTION-CHA}$

$SE(\mathcal{K}, \Pi', \mathcal{X})$ compute the **Maximally ContainD** position in all computed conflicts. Each position p is attributed a rank that indicates the number of conflicts containing p . The question is generated on the position that has the maximum rank. If more than one position are attributed the same maximum rank, one is picked randomly. Using CH, the maximally contained position p corresponds to the vertex of maximum degree. Obviously, this strategy can avoid asking unnecessary questions by looking ahead and choosing the position that resolves as many conflicts as possible.

Algorithm 4 Inquiry strategy

```

1: function INQUIRY( $\mathcal{K}, \Pi$ )
2:    $\mathcal{K}' \leftarrow \mathcal{K}$ 
3:    $\Pi' \leftarrow \Pi$ 
4:    $C_{naive} = \text{allconflicts}_{naive}(\mathcal{K}')$ 
5:   /* Start phase one */
6:   while  $C_{naive} \neq \emptyset$  do
7:      $\phi \leftarrow \text{GENERATEQUESTION}(\mathcal{K}, \Pi', \mathcal{X})$ 
8:      $f \leftarrow \text{ASKUSER}(\phi)$ 
9:      $\mathcal{K}' \leftarrow (\text{apply}(\mathcal{F}, f), \Sigma_T, \Sigma_C)$ 
10:     $\Pi' \leftarrow \Pi' \cup \{(A, i) \mid f = (A, i, t)\}$ 
11:     $\text{UPDATECONFLICTS}(\mathcal{K}')$ 
12:  end while
13:  /* Start phase two */
14:  while  $\text{CHECKCONSISTENCY-OPT}(\mathcal{K}') = \text{false}$  do
15:     $\phi \leftarrow \text{GENERATEQUESTION-CHASE}(\mathcal{K}, \Pi', \mathcal{X})$ 
16:     $f \leftarrow \text{ASKUSER}(\phi)$ 
17:     $\mathcal{K}' \leftarrow (\text{apply}(\mathcal{F}, f), \Sigma_T, \Sigma_C)$ 
18:     $\Pi' \leftarrow \Pi' \cup \{(A, i) \mid f = (A, i, t)\}$ 
19:  end while
20:  return  $\mathcal{K}'$ 

```

Algorithm 5 Sound questions for a strategy

```

1: function SOUNDQUESTION( $\mathcal{K}, \Pi, \mathcal{X}$ )
2:    $\mathcal{X} = (N, h)$ 
3:    $\phi \leftarrow \emptyset$ 
4:    $\Pi' \leftarrow \text{RETRIEVE-POSITIONS}(\mathcal{X}, \mathcal{K})$ 
5:   for each  $(A, j) \in \Pi' \setminus \Pi$  do
6:      $val \leftarrow \text{adom}(A, i, \mathcal{F}) \setminus \{\text{value}_A^i(\mathcal{F})\}$ 
7:      $val \leftarrow val \cup \{X_A^i\}$ 
8:      $\phi \leftarrow \phi \cup \{(A, i, t) \mid t \in val\}$ 
9:   end for
10:  for each  $f_k = (A, i, t) \in \phi$  do
11:     $\Pi_{tmp} \leftarrow \Pi \cup \{(A, i)\}$ 
12:     $\mathcal{K}' \leftarrow (\text{apply}(\mathcal{F}, f), \Sigma_T, \Sigma_C)$ 
13:    if  $\Pi\text{-REPOPT}(\mathcal{K}', \Pi'_{tmp}) = \text{false}$  then
14:       $\phi \leftarrow \phi \setminus (A, i, t)$ 
15:    end if
16:  end for
17:  return  $\phi$ 

```

6 EXPERIMENTAL STUDY

Our experimental assessment is devoted to study two major features of our user-guided repairing framework: (i) *effectiveness*: investigates to what extent the framework is efficient in helping the user repair the knowledge base with minimal effort, in terms of average number of asked questions per strategy and average number of conflicts per question, and (ii) *delay time*: the efficiency of our framework when it comes to maintaining a reasonable delay time between each asked question. By a reasonable delay time we intend a delay less than 1 to 2 seconds as discussed in [26].

Experimental setup. We have implemented our framework using Java 1.8 on a 2.40GHz 4-core, 16Gb laptop running Windows 7.

We have used GRAAL⁴ as a chase engine. Each experiment has been repeated a number of times, as indicated in the individual plots (after discarding the cold start). As there are no existing datasets or benchmarks equipped with the rich set of constraints we consider in this paper, we rely on synthetically generated knowledge bases and corresponding constraints. We also employ a real-world knowledge base on Durum Wheat from [2]. This knowledge base has been constructed manually from documents and reports, which led to have notable inconsistencies. Moreover, the attached constraints (including TGDs and CDDs) have been validated by experts. Such a KB turned to be suitable for our experiments as it fits the assumption that the set of facts is dirty and the set of constraints is reliable.⁵

Synthetic KBs. The synthetic knowledge bases were generated by tuning some input parameters. We first generate a *vocabulary* of the knowledge base, i.e. predicate, variable, and constant spaces by allowing also n-ary relations. Each predicate is assigned a random arity from 2 to 10 following a uniform probability distribution. A given number of CDDs are generated over the vocabulary by parameterizing the number of atoms s involved in the CDDs and the percentage $v\%$ of atoms positions corresponding to join variables such that $s \in [5, 10]$ and $v\% \in [10\%, 100\%]$. TGDs are generated following the same procedure as CDDs. To make the knowledge base meaningful, links between TGDs and CDDs are made so that some TGDs may introduce facts that will violate the CDDs. A *depth* $d_{\mathcal{K}}$ for a given knowledge base \mathcal{K} is defined as how many TGDs applications are needed to violate a CDD. A conflict depth $d_{\mathcal{K}} = 2$ means for each CDD we need the application of two distinct TGDs so that the CDD is violated. Inconsistent KBs are generated as follows, for a given facts size $n_{\mathcal{F}}$ and an *inconsistency ratio* r_{inc} ⁶, we keep generating sets of atoms that violate the CDDs until we reach r_{inc} . Then, we pad the set of facts \mathcal{F} with atoms that are not involved in any conflicts. We add two indicators of the structure of conflicts in the KB, namely “Avg # atoms per overlap” and “Avg scope”. The first embodies the average number of atoms in each overlap, an overlap being the intersection between at least two conflicts. The avg scope indicates for each conflict how many conflicts are overlapping with it, this number being averaged over the total number of conflicts.

The Durum Wheat KBs. The real-world Durum Wheat knowledge base in our experiments has been augmented with new domain-specific TGDs and CDDs. The table in Figure 2 presents the different characteristic of the knowledge bases and an example of facts, a TGD and a CDD. Please note that ChaseSize (#atoms) refers to the size of the facts after applying the chase. We made two versions of the knowledge base of increasing size of CDDs, i.e. Durum Wheat v1 and Durum Wheat v2. Notice that the number of conflicts increases from v1 to v2 while inconsistency ratio stays the same. This is due to the fact that the conflicts newly discovered by the added constraints in v2 involve the same number of atoms.

We have simulated the end-user via an algorithm that randomly chooses a valid fix from the proposed fixes following a uniform probability distribution. Considering other kinds of distributions and, in particular, choosing the most appropriate probability distribution that can simulate all the user’s choices is not trivial and falls under user modeling, which is beyond the scope of our paper.

⁴<http://graphik-team.github.io/graal/>.

⁵Notice that we could not use popular knowledge bases such as YAGO, DBPedia and LUBM because of their limited expressiveness on the vocabulary (only binary relations) and lack of TGDs and CDDs.

⁶Number of atoms involved in at least one conflict divided by $n_{\mathcal{F}}$.

Analysis of Durum Wheat KBs. We measure the average number of asked questions for each strategy in order to gauge the effectiveness of our approach on our real-world dataset. Figure 2 (a) and (b) show the results for all the considered strategies. We can observe that *opti-mcd* is outperforming the other strategies on Durum Wheat v1 with an average of 14.18 questions asked. This difference is also observed on Durum Wheat v2 in Figure (a) where *opti-mcd* outperforms other strategies with an average of 29.36 questions asked. The reason why *opti-mcd* is the winning strategy is due to the fact that this strategy is actually capable of exploiting the overlapping among conflicts which is given by the indicator *avg scope*. In this case, the value of such indicator is 8 meaning that in the best case, roughly speaking, each question can solve 8.1 (or 7.1 for V2) conflicts. This is under the assumption that the conflicts are overlapping on the same atoms. Regarding the difference with the other strategies, the results show that the strategies (other than *opti-mcd*) tend to behave the same as they do not exploit such a property. In addition, notice that *opti-join* and *opti-prop* are very close to random strategy. This is due to the fact that the percentage of join positions in conflicts is close to 90%. This makes the probability of choosing a join position with random strategy very high. Moreover, the increase in the average number of asked questions in all strategies in v2 is explained by the fact that v2 has slightly more conflicts than v1.

Another perspective that gives a better illustration of the effectiveness of our interactive strategies is the average number of resolved conflicts per question in Figures (c) and (d). The former is computed as total nr. of conflicts/total nr. of questions (per strategy). Again, we can observe that the *opti-mcd* strategy handles more conflicts per question on average and proves to be the most effective strategy compared to the others.

Analysis of Synthetic KBs. We now analyze the effectiveness where the average number of asked questions is measured for each strategy on synthetically generated KBs. For the first experiment described in Figure 3, we generated a knowledge base with only CDDs and no TGDs. Then, we increased the inconsistency ratio by increments of 5% while keeping the size of the knowledge base fixed (see the table in Figure 3 for characteristics). The results show a good performance of *opti-mcd*, while *opti-join* and *opti-prop* behave similarly. The random strategy performs the worst among them. This result in fact confirms the observation already made on the Durum Wheat knowledge base about overlapping conflicts. In this experiment, we notice a larger gap between *opti-join* and *opti-prop* strategies on one side and random strategy on the other side. Since the percentage of join positions in conflicts is very low in the generated atoms (under to 30%), it is less likely that the random strategy would randomly choose a join position. The average number of resolved conflicts per question is shown in Figure 3 (b) where one can see the performance of each strategy.

We should highlight that the baseline strategy (*random*) in the two experiments is performing quite well as it still asks less questions than the number of conflicts. This is quite natural as the conflicts are in fact overlapping (as confirmed by the two indicators, *avg # atoms per overlap* and *avg scope*) hence many conflicts are resolved via the resolution of other conflicts. However, there is a huge gap between the performance of the baseline strategy and those of the more optimized strategies.

The second experiment aims at studying the convergence of each strategy. Figure 4 (a) is done on an inconsistent knowledge base with CDDs and no TGDs. We can observe that as the strategies proceed with questioning, they exhibit different speeds in getting toward a full resolution of the conflicts. While *opti-mcd*

is faster than all the other strategies, *opti-join* and *opti-prop* are quite similar with a small difference on the number of asked questions. Figure 4 (b) is done on a fixed inconsistent knowledge base with both CDDs and TGDs. We can observe that each strategy hits the lowest number of conflicts at a point (close to 0) then it starts slightly fluctuating until convergence. The rapid descending phase corresponds to the process of handling only CDDs that are directly violated by the initial set of facts without taking into account the TGDs. Once the TGDs are triggered and the chase starts, it interleaves TGDs with CDDs, leading to up and down fluctuations corresponding to new conflicts introduced by TGDs and resolution of conflicts with CDDs, respectively. Continuous lines between fluctuations correspond to stagnation, in which neither the questions are resolving conflicts nor triggering TGDs and CDDs brings new conflicts. The strategies *opti-mcd*, *opti-join* and *opti-prop* behave quite similarly with a notable difference in the convergence speed, bringing *opti-mcd* to be the fastest.

The next experiment is devoted to measure the delay time between questions for synthetic KBs. Note that the delay time for all previous experiments was very reasonable (less than 0.2 seconds) for both synthetic and Durum Wheat knowledge bases.

Figure 5 shows three different measures of the delay time using *opti-mcd* in all experiments. The delay time for the other strategies had a similar trend and is omitted for space reasons.

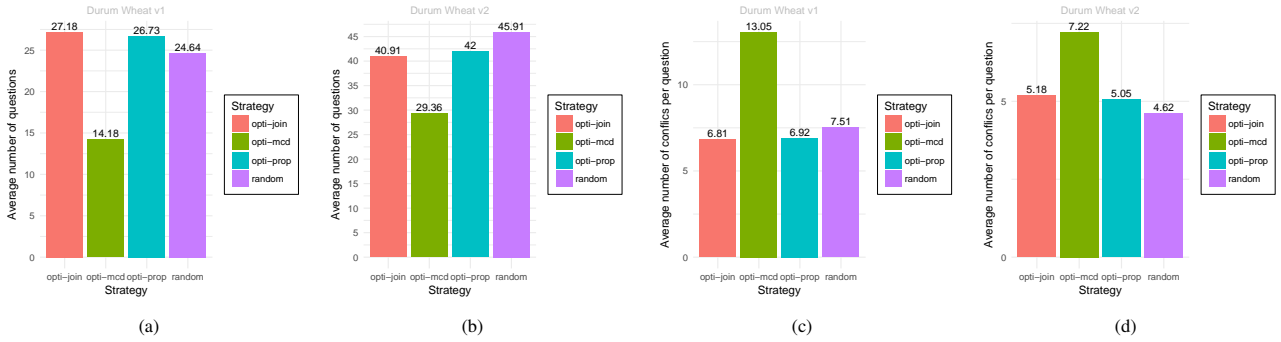
The goal of the experiment whose results are reported in Figure 5 (a) is to investigate whether increasing inconsistency (from 20% to 80%) would affect the delay time. We can observe that the inconsistency ratio is rather independent from the delay time. This is quite interesting as regardless of the inconsistency degree of the knowledge base, interactivity is guaranteed with the user in a very reasonable time (average is less than 0.25 sec). Some outliers are highlighted in the boxplot, however they stay within the limits of reasonable delay time (less than 0.8 sec).

In the next experiment (Figure 5 (b)), we employed a KB of increasing size (up to 20%, 40% and 60%), respectively while keeping the inconsistency ratio fixed to 30%. The delay time grows as the size of the knowledge base grows. Moreover, the boxplot shows that the variance of delay time increases as the size of the KB increases. This result shows that our method needs a piecemeal application of interactive repairing and can always be applied to small portions of the KB.

In the next experiment, we have chosen a worst-case scenario in which we have a fully inconsistent KB, corresponding to inconsistency ratio of 100%, and we vary the depth (from d_1 to d_4) of the dependencies involved (both TGDs and CDDs). For all depth d_i we have $\#CDD(d_i) = 150$, and $\#TGDs(d_1) = 50$, $\#TGDs(d_2) = 100$, $\#TGDs(d_3) = 150$, $\#TGDs(d_4) = 200$. We have already shown in the experiment of Figure 5 (a) that increasing the inconsistency ratio does not affect the delay time. We observe that the delay time increases with depth, in fact the larger the depth the more time the chase takes while repairing. Notice that the chase is involved in computing Π -repairability and consistency check. Overall, the delay time is kept low for all depths and less than 2 seconds.

7 CONCLUSION AND FUTURE WORK

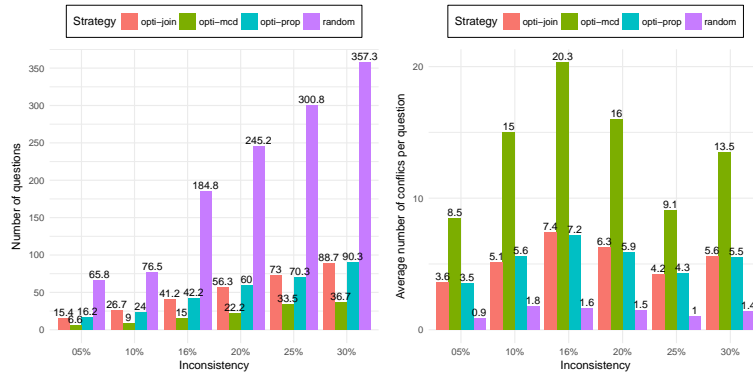
In this paper, we have presented a novel user-guided repairing technique for knowledge bases, leveraging updates and interplay of dependencies (TGDs and CDDs). Several extensions can be thought of, such as formalization of user modeling to represent several classes of users (from domain experts to non-experts), and learning from provided user choices in the questioning strategies.



KB	Size (#atoms)	ChaseSize (#atoms)	Conflicts	Avg # atoms per overlap	Avg scope	#Repetitions
DURUM WHEAT V1	567	1075	185	1.42	8.1	10
DURUM WHEAT V2	567	1075	212	1.41	7.8	10
KB	#TGDs	#CDDs	Inconsistency ratio	Avg # atoms per conflict		
DURUM WHEAT V1	269	27	14% (79 atoms)	3		
DURUM WHEAT V2	269	100	14% (79 atoms)	2		

Durum Wheat KB	Content
\mathcal{F}	<i>hasPrecedent(soil2, vacoparis), sorghum(vacoparis), soil(soil2).</i> soil2 has a precedent vacoparis of type sorghum.
Σ_T	<i>isCultivatedOn(X1, X2), durum_wheat(X1), soil(X2) \rightarrow hasPrecedent(X2, X3), soybean(X3)</i> if a durum wheat is cultivated on a soil then the precedent on this soil is soybean.
Σ_C	<i>isAtGrowingStage(X, Z), isPerformedOn(X1, X), tillering_begins(Z), durum_wheat(X), fertilization(X1) $\rightarrow \perp$</i> it is forbidden (or impossible) to apply a fertilization on a durum wheat if it is in the beginning of the tillering growth stage.

Figure 2: Average number of questions per strategy on the Durum Wheat knowledge bases.



(a) Fixed size KB (1005 atoms) with increasing inconsistency ratio. CDDs and no TGDs.

(b) Average conflicts per question of (a).

KB	Size (#atoms)	ChaseSize (#atoms)	Conflicts	Avg # atoms per overlap	Avg scope	#Repetitions
“05%”	1005	1005	56	1.45	9.8	6
“10%”	1005	1005	135	1.57	12.8	6
“16%”	1005	1005	304	1.68	33.9	6
“20%”	1005	1005	356	1.65	30.5	6
“25%”	1005	1005	304	1.67	34.5	6
“30%”	1005	1005	496	1.66	31.6	6

Figure 3: Average number of questions per strategy on synthetic knowledge bases. The percentage on the axes represent inconsistency ratio.

We also believe that more challenges may arise in extending this work to full-fledged denial constraints and arbitrary (non weakly acyclic) TGDs.

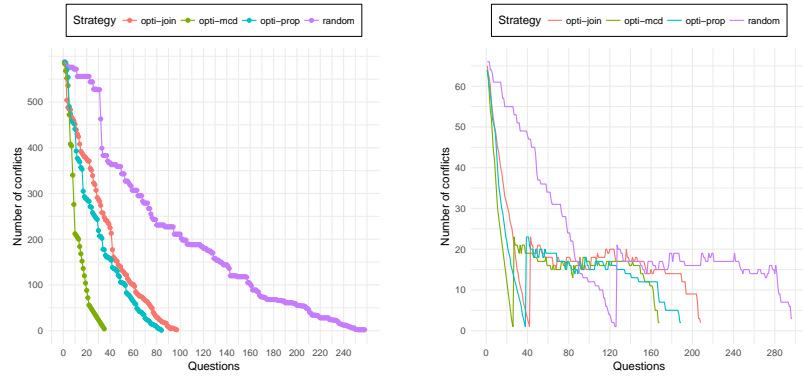
REFERENCES

[1] Marcelo Arenas, Leopoldo Bertossi, and Jan Chomicki. 1999. Consistent query answers in inconsistent databases. In *Proc of SIGMOD*. ACM, 68–79.

[2] Abdallah Arioua, Patrice Buche, and Madalina Croitoru. 2016. A Datalog \pm Domain-Specific Durum Wheat Knowledge Base. In *Proc. of MTSR 2016*. Springer, 132–143.

[3] Sebastian Arming, Reinhard Pichler, and Emanuel Sallinger. 2016. Complexity of Repair Checking and Consistent Query Answering. In *LIPICs-Leibniz International Proceedings in Informatics*, Vol. 48, 21:1–21:18.

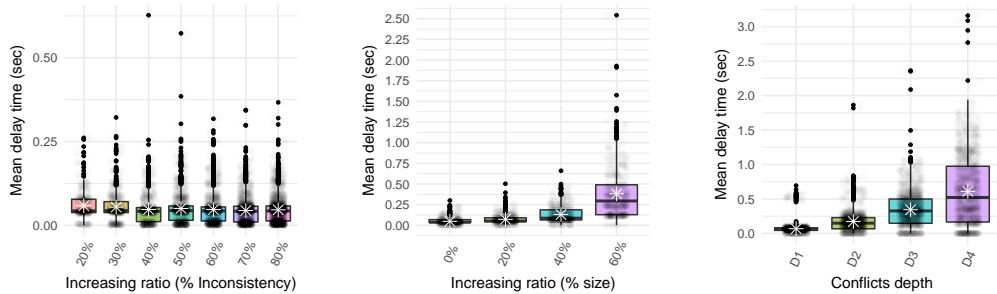
[4] Ahmad Assadi, Tova Milo, and Slava Novgorodov. 2017. DANCE: Data Cleaning with Constraints and Experts. In *Proc. of ICDE*. 1409–1410.



(a) Fixed size KB (3004 atoms) with constant inconsistency ratio 25%. With only CDDs and no TGDs.

(b) Fixed size KB (800 atoms) with constant inconsistency ratio of 25%, 50 CDDs and 25 TGDs. Total number of conflicts after applying the chase is 136.

Figure 4: The convergence of strategies over a question/answer session.



(a) Fixed size KB (3000 atoms) with increasing inconsistency ratio.

(b) Increasing size KB starting at 0% with 3000 atoms, constant inconsistency ratio 30%.

(c) Fixed size KB (400 atoms) with constant inconsistency ratio of 100%, varied depth (1 to 4).

Figure 5: Average delay time with 5 repetitions for each percentage. (c) is the delay time when TGDs and CDDs are considered, $\#CDD(d_1) = 150$, and $\#TGDs(d_1) = 50$, $\#TGDs(d_2) = 100$, $\#TGDs(d_3) = 150$, $\#TGDs(d_4) = 200$. *Opti-mcd* strategy is used. Asterix represents the mean.

- [5] Jean-François Baget, Michel Leclère, Marie-Laure Mugnier, and Eric Salvat. 2011. On rules with existential variables: Walking the decidability line. *Artificial Intelligence* 175, 9-10 (2011), 1620–1654.
- [6] Michael Benedikt, George Konstantinidis, Giansalvatore Mecca, Boris Motik, Paolo Papotti, Donatello Santoro, and Efthymia Tsamoura. 2017. Benchmarking the Chase. In *Proc. of PODS*. 37–52.
- [7] Leopoldo Bertossi. 2011. Database repairing and consistent query answering. *Synthesis Lectures on Data Management* 3, 5 (2011), 1–121.
- [8] Sourav S. Bhowmick, Byron Choi, and Chengkai Li. 2017. Graph Querying Meets HCI: State of the Art and Future Directions. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. 1731–1736.
- [9] Philip Bohannon, Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. 2007. Conditional Functional Dependencies for Data Cleaning. In *Proc. of ICDE*. 746–755.
- [10] Philip Bohannon, Michael Flaster, Wenfei Fan, and Rajeev Rastogi. 2005. A Cost-Based Model and Effective Heuristic for Repairing Constraints by Value Modification. In *Proc. of SIGMOD*. 143–154.
- [11] Andrea Cali, Georg Gottlob, and Thomas Lukasiewicz. 2012. A general Datalog-based framework for tractable query answering over ontologies. *Journal of Web Semantics* 14 (2012), 57–83.
- [12] Andrea Cali, Georg Gottlob, and Andreas Pieris. 2012. Towards more expressive ontology languages: The query answering problem. *Artificial Intelligence* 193 (2012), 87–128.
- [13] Xu Chu, Ihab F Ilyas, and Paolo Papotti. 2013. Holistic data cleaning: Putting violations into context. In *Proc. of ICDE*. IEEE, 458–469.
- [14] Xu Chu, John Morcos, Ihab F. Ilyas, Mourad Ouzzani, Paolo Papotti, Nan Tang, and Yin Ye. 2015. KATARA: A Data Cleaning System Powered by Knowledge Bases and Crowdsourcing. In *Proc. of SIGMOD*. 1247–1261.
- [15] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. 2005. Data exchange: semantics and query answering. *Theor. Comput. Sci.* 336, 1 (2005), 89–124.
- [16] Wenfei Fan and Ping Lu. 2017. Dependencies for Graphs. In *Proc. of PODS*. 403–416.
- [17] Wenfei Fan, Yinghui Wu, and Jingbo Xu. 2016. Functional Dependencies for Graphs. In *Proc. of SIGMOD*. 1843–1857.
- [18] Mina H. Farid, Alexandra Roatis, Ihab F. Ilyas, Hella-Franziska Hoffmann, and Xu Chu. 2016. CLAMS: Bringing Quality to Data Lakes. In *Proc. of SIGMOD*. 2089–2092.
- [19] Floris Geerts, Giansalvatore Mecca, Paolo Papotti, and Donatello Santoro. 2013. The LLUNATIC Data-Cleaning Framework. *PVLDB* 6, 9 (2013), 625–636.
- [20] Víctor Gutiérrez-Basulto, Yazmín Ibáñez García, Roman Kontchakov, and Egor V. Kostylev. 2015. Queries with Negation and Inequalities over Lightweight Ontologies. *Web Semant.* 35, P4 (Dec. 2015), 184–202.
- [21] Jian He, Enzo Veltri, Donatello Santoro, Guoliang Li, Giansalvatore Mecca, Paolo Papotti, and Nan Tang. 2016. Interactive and Deterministic Data Cleaning. In *Proc. of SIGMOD*. 893–907.
- [22] Neil Immerman. 1989. Expressibility and parallel complexity. *SIAM J. Comput.* 18, 3 (1989), 625–638.
- [23] David S. Johnson, Mihalis Yannakakis, and Christos H. Papadimitriou. 1988. On generating all maximal independent sets. *Inform. Process. Lett.* 27, 3 (1988), 119–123.
- [24] Solmaz Kolahi and Laks VS Lakshmanan. 2009. On approximating optimum repairs for functional dependency violations. In *Proceedings of the 12th International Conference on Database Theory*. ACM, 53–62.
- [25] Domenico Lembo, Maurizio Lenzerini, Riccardo Rosati, Marco Ruzzi, and Domenico Fabio Savo. 2010. Inconsistency-tolerant Semantics for Description Logics. In *Proceedings of the International Conference on Web Reasoning and Rule Systems (RR'10)*. Springer-Verlag, 103–117.
- [26] Robert B. Miller. 1968. Response Time in Man-computer Conversational Transactions. In *Proc. of FJCC 1968*. ACM, 267–277.
- [27] Nataliya Prokoshyna, Jaroslaw Szlichta, Fei Chiang, Renée J. Miller, and Divesh Srivastava. 2015. Combining Quantitative and Logical Data Cleaning. *PVLDB* 9, 4 (2015), 300–311.
- [28] Jef Wijsen. 2005. Database repairing using updates. *ACM TODS* 30, 3 (2005), 722–768.
- [29] Mohamed Yakout, Ahmed K. Elmagarmid, Jennifer Neville, Mourad Ouzzani, and Ihab F. Ilyas. 2011. Guided data repair. *PVLDB* 4, 5 (2011), 279–289.

Synchronous Multi-GPU Deep Learning with Low-Precision Communication: An Experimental Study

Demjan Grubic
 Department of Computer Science
 ETH Zurich
 dgrubic@student.ethz.ch

Leo Tam
 NVIDIA
 leot@nvidia.com

Dan Alistarh
 IST Austria
 dan.alistarh@ist.ac.at

Ce Zhang
 Department of Computer Science
 ETH Zurich
 ce.zhang@inf.ethz.ch

ABSTRACT

Training deep learning models has received tremendous research interest recently. In particular, there has been intensive research on reducing the *communication cost* of training when using multiple computational devices, through reducing the precision of the underlying data representation. Naturally, such methods induce system trade-offs—lowering communication precision could decrease communication overheads and improve scalability; but, on the other hand, it can also reduce the accuracy of training. In this paper, we study this trade-off space, and ask: *Can low-precision communication consistently improve the end-to-end performance of training modern neural networks, with no accuracy loss?*

From the performance point of view, the answer to this question may appear deceptively easy: compressing communication through low precision should help when the ratio between communication and computation is high. However, this answer is less straightforward when we try to generalize this principle across various neural network architectures (e.g., AlexNet vs. ResNet), number of GPUs (e.g., 2 vs. 8 GPUs), machine configurations (e.g., EC2 instances vs. NVIDIA DGX-1), communication primitives (e.g., MPI vs. NCCL), and even different GPU architectures (e.g., Kepler vs. Pascal). Currently, it is not clear how a realistic realization of all these factors maps to the speed up provided by low-precision communication. In this paper, we conduct an empirical study to answer this question and report the insights.

1 INTRODUCTION

The system tradeoffs induced by training deep neural networks seem to be an endless discussion [9, 11, 15, 19, 24, 34, 46]. One reason for this sophistication is the level of diversity involved. At the algorithmic level, there are synchronous, asynchronous, and hybrid approaches. At the workload level, different neural networks have different computation and data movement patterns. At the architecture level, different computation devices, such as CPUs, GPUs, and FPGAs offer sharply different tradeoffs. Making matters worse, these are not pure performance tradeoffs, which one could tackle with performance modeling. Instead, different point in the tradeoff space has different *accuracy* of the trained model, a property that is very difficult to predict, and for which has there currently exists little theoretical understanding [49].

A Data Management Angle. From a data management perspective, these trade-offs provide an opportunity to build a *automatic optimizers* for deep learning tasks. Just as with previous optimizers that our community has been building [6, 8], the understanding of the tradeoff is often the prerequisite. Given the limited current theoretical understanding of deep learning, such modeling is inevitably empirical for the immediate future. Therefore, we believe that there is timely necessity for a set of empirical, fair, comparison to reveal the tradeoffs behind building and optimizing deep learning systems.

In this paper, we present an in-depth empirical study which focuses on a *subspace* of the whole tradeoff, that is, the tradeoff introduced by *the precision of communication when training deep neural networks with a synchronous multi-GPUs system*.

Low Precision Deep Learning. An emerging topic in deep learning systems is lowering the precision of data representation throughout the whole system [5, 14, 18, 20, 23, 33, 36, 50, 51]. There has been recent success in representing *all* data movements involved in training a neural network with as little as 1-bit per dimension while still getting the same accuracy for *some, but not all, networks* [36]. However, none of these work depict a full tradeoff space — they either focus on extreme cases (e.g., 1-bit) with significant loss of accuracy and do not discuss the impact of adding more bits, or only focus on algorithmic aspects, without a well-implemented system. In this paper, we conduct an empirical study to understand the impact of lower precision of data representation both on the training accuracy and the speed.

System Artefact. A fair empirical study calls for a system that achieves optimized performance for each configuration in the above space. Surprisingly, such a system does not exist off-the-shelf: if we just take CNTK or TensorFlow, the performance of many configurations is not as optimized as they could be. Thus, we decide to start from CNTK but conduct intensive performance optimization to ensure the fairness of our study. We first developed a system prototype optimized for low-precision machine learning. In this paper, we report an array of system optimizations that leads to up to 3.5× speed up over CNTK, which allows us to understand the tradeoff as fair as possible.

Summary of Contribution 1: Experimental Study. We consider the task of training deep neural networks on a single system with multiple GPUs, in a setting where all these GPUs communicate in a synchronous manner [24, 34]. In this setting, our first contribution is a study of the impact of *varying the number of bits to communicate between these devices from 1bit to 32bits*. Our study contains the following axes in the tradeoff space:

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

- (1) **Machine Learning Tasks/Datasets:** {Image, Speech}
- (2) **Neural Network Architectures:** {AlexNet [27], VGG [40], ResNet [21], Inception [43], LSTM [22]}
- (3) **Quantization Strategy:** {"1-bit SGD" [39], QSGD [5]}
- (4) **Number of GPUs:** {1, 2, 4, 8, 16}
- (5) **Type of Machines:** {Amazon EC2, NVIDIA DGX-1}
- (6) **Programming Models:** {MPI, NCCL}
- (7) **GPU Architectures:** {Kepler, Pascal}

The tradeoff space we studied contains a full cross products the above axes (whenever possible); for each configuration, we vary the precision of data communication and measure (1) end-to-end performance (seconds), (2) convergence speed (#iterations), (3) speed per iteration (seconds), and (4) accuracy (%).

To fully depict this tradeoff, our study used more than 1400 machine hours on Amazon EC2's recently released 16-GPU instance and more than 20 machine hours on NVIDIA DGX-1. These give us an overview of how recent hardware and software have impact on the importance of low precision communication. This is so far the most comprehensive study of the impact of low-precision communication to deep learning. The system insights we get go significantly beyond those of previous work.

Summary of Contribution 2: Insights. Our study not only reveals insights beyond those of previous work, but also provides the first comparable quantitative evaluation of the results scattered in previous work that was original evaluated on different platforms and settings. We now briefly summarize these insights.

1. Does low-precision always hurt accuracy?

No. Across all datasets and models that we evaluated, we found parameter settings under which low-precision variants are able to achieve the same accuracy as full precision. Specifically, on all networks we evaluated, when quantizing communication using the 1bitSGD algorithm, a low-precision system is able to achieve the same accuracy (within 0.2%) as a full precision run. Using QSGD usually requires slightly higher precision: using 4-bit gradients always preserves the same accuracy (within 0.1%), while the 8-bit variant always matches or even *improves* final accuracy (within 0.5%). We note a few caveats regarding this result:

- (1) Different layers have different sensitivity to quantization: for convolutional neural networks, the convolutional layers require more bits than the fully connected layers.
- (2) Quantizing too aggressively can lead to significant accuracy loss: we identify scenarios where 2bit QSGD can no longer train to state-of-the-art precision.

2. Does low-precision always help performance?

Not always. The answer to this question depends on several factors. We postpone a detailed discussion to Section 5.2.

- (1) The most surprising result regarding performance regards the impact of the communication primitives used. When we replace MPI with the NCCL, a restricted set of communication primitives optimized by NVIDIA for GPU-to-GPU communication, 32bit full precision becomes much faster than MPI. Consequently, the speedup we can obtain with low-precision communication is also limited. In fact, on most networks, the performance improvement we get when using 8GPUs and NCLL is almost negligible; only for one network (VGG), we only get up to 1.4× speedup.
- (2) With slower MPI (which can be seen as a proxy for a slower interconnect), the tradeoff becomes more significant. One key factor is the ratio between computation

(time to calculate the gradient) and communication (time to communicate the gradient). For networks with a large number of parameters such as AlexNet, we observe up to 4× speedup using low-precision communication; For networks with small model, we observe almost no speedup.

- (3) The number of GPUs also plays a role. Naturally, when the number of GPUs increases, the necessity of using low-precision communication increases.

3. Is using extremely low precision ever helpful?

Rarely. We observe a trend of “diminishing returns” when lowering the precision of gradient to extreme cases such as a single bit, while such compression can lead to accuracy loss. Examining the time cost of a dataset iteration, even with MPI, using 1-bit rarely outperforms using 4-bit on our benchmarks. Through simulation, we can project a communication-to-computation regime where extreme low precision would have significant impact, but none of the existing networks fall into that regime.

4. Have the current programming models unleashed the full potential of low-precision machine learning?

No. NCCL hardcodes the reduction semantics with 32-bit full precision, and only offers a limited set of binary operations. Hence, there are currently no easy ways to use the NCCL primitives to support low-precision reduction efficiently. We conduct a simulation which implies that a version of NCCL with low-precision support could lead to a low-precision system that is up to 1.4× faster than our current prototype. MPI is also cumbersome to use in the context of low precision, although its richer semantics allow us to implement efficient variants of quantized methods.

5. Do we really need 16 GPUs on a single instance?

Rarely when training a single model. Amazon EC2 provides a single instance with 16 GPUs: p2.16xlarge, whose price is 2× higher than a p2.8xlarge 8-GPU instance, and currently stands at \$14/hour. We only find few scenarios where the speedup achieved with 16 GPUs versus 8 would justify the cost doubling.

Limitations. Our study has the following limitations.

The first is that our study assumes that the user has *zero error-tolerance*. That is, we focus on scenarios where the user wishes to obtain a model with the same accuracy as one trained at full precision, but wishes to save time (and money) on training by using quantized methods. If the user could tolerate higher errors (e.g., 10% accuracy loss), the results of our study might change, as we should be able to use more aggressive low-precision schemes.

The second limitation is that our study only considers the communication overhead when exchanging gradients across devices. Other potential benefits of using low-precision data representation is to speed up computation (each register can hold more numbers). We do not consider these, as the current GPU platform does not yet provide enough flexibility (e.g., 4-bit computation). A related limitation is that we only focus on GPU platforms. The current work is part of a larger ongoing study examining computation-to-communication trade-offs across CPU, GPU, XEON PHI, and FPGA; however, these other platforms are currently out of the scope of a single paper.

The third is that our study focuses on the speed of *training*, which is currently the most computationally-intensive aspect of neural networks. When the objective changes, e.g., energy efficiency, or speed of inference, the results might also change.

The fourth limitation is that we build our artefact starting from a specific platform, Microsoft CNTK. However, we expect similar observations to hold in the context of other deep learning systems, such as TensorFlow. All these systems use the same computational substrate (NVIDIA CUDA and cuDNN libraries [10]), and the semantics of the allreduce-based gradient exchange are similar across all synchronous systems. However, we do not wade into an in-depth quantitative analysis to support this claim. In the future, we plan to extend our study to a variety of systems, but this is out of the scope of this single paper.

The last limitation is that our study makes inherent assumptions of how different hyperparameters are tuned, e.g., learning rate and batch size. Although we believe our protocol of hyperparameters is reasonable and fair, it is also true that we did not fully explore all possible combinations of these hyperparameters, as the cost would increase significantly: by a rough estimate, an exhaustive search of the hyperparameter space would cost more than \$1M on EC2.

Reproducibility. All of our experiments can be reproduced using a virtual machine available on EC2: `i-0a7a0521e93c329d9`. Our variant of CNTK is available under `github.com/ZipML/CNTK`.

Overview. The rest of this paper is organized as follows. We introduce the background material in Section 2 and describe our system prototype in Section 3. We describe our experiment setup in Section 4 and analyze our results in Section 5. We provide more discussion in Section 5.4 and survey related work in Section 7.

2 PRELIMINARIES

We present background for our study. We first describe the stochastic gradient descent (SGD) algorithm and the synchronous parallel version of it which is the focus of our study. We then describe two low-precision SGD algorithms and discuss two different communication primitives based on MPI and NCCL.

2.1 Stochastic Gradient Descent

Stochastic gradient descent (SGD) is the workhorse algorithm in training deep learning models. SGD is developed for a more general class of optimization problems: Let $f : \mathcal{R}^n \rightarrow \mathcal{R}$, it solves $\min_{\mathbf{x}} f(\mathbf{x})$. The assumption of SGD is that we have access to the *stochastic gradients* of this function. We denote the stochastic gradient by \tilde{g} which satisfies $E[\tilde{g}(\mathbf{x})] = \nabla f(\mathbf{x})$. SGD will converge towards the minimum by iterating the following procedure

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \eta_t \tilde{g}(\mathbf{x}_t),$$

where \mathbf{x}_t is the current state of the model, and η_t , also called the *step-size* or *learning rate* is a hyperparameter.

In the machine learning setting, we are given i.i.d. data points X_1, \dots, X_m generated from an unknown distribution D , and a loss function $\ell(X, \theta)$, which measures the loss of the model θ at data point X . We wish to find a model θ^* which minimizes $f(\theta) = E_{X \sim D}[\ell(X, \theta)]$, the expected loss to the data. Since for each i , the function $\nabla \ell(X_i, \theta)$ is a stochastic gradient for f , we can use SGD to find θ^* . This captures neural network training.

Synchronous Parallel SGD. In this paper, we focus on synchronous *data-parallel* SGD, modeling multi-GPU systems in which each GPU has a complete copy of the model. GPUs proceed in synchronous steps, and communicate using direct messages. Each of the K processors maintains a local copy of the model \mathbf{x} , of dimension n . The algorithm is described in Algorithm 1.

Each processor aggregates the value of \mathbf{x} , then obtains random gradient updates for each component of \mathbf{x} , then communicates

Data: Stochastic gradients

Data: Local copy of the parameter vector \mathbf{x}

```

1 for each iteration  $t$  do
2   Let  $\tilde{g}_t^i$  be an independent stochastic gradient ;
3    $M^i \leftarrow \text{Encode}(\tilde{g}_t^i(\mathbf{x}))$  //encode gradients ;
4   broadcast  $M^i$  to all peers;
5   for each peer  $\ell$  do
6     receive  $M^\ell$  from peer  $\ell$ ;
7      $\tilde{g}^\ell \leftarrow \text{Decode}(M^\ell)$  //decode gradients ;
8   end
9 end

```

Algorithm 1: Synchronous Data-Parallel SGD Algorithm.

Data: Gradient vector \mathbf{v} to be quantized

Data: Error ϵ from the previous round

```

1  $\mathbf{v} \leftarrow \mathbf{v} + \epsilon$  //add error from previous round ;
2 for each component  $i$  do
3    $q_i \leftarrow \text{avg}_+$  if  $v_i \geq 0$ ,  $\text{avg}_-$  otherwise;
4    $\epsilon_i \leftarrow v_i - q_i$ ;
5 end
6 return  $\mathbf{q}$ 

```

Algorithm 2: 1bitSGD procedure.

these updates to all other nodes, and finally aggregates the received updates and applies them to its local model. We have added *encoding* and *decoding* steps for the gradients before and after send/receive in lines 1 and 7. In the following, we assume the above pattern. Whenever describing a communication-efficient variant of SGD, we only specify the *encode/decode* functions. Note that these encoding process is often *lossy* that only recover an *approximate* gradient.

When the encoding and decoding steps are trivial (i.e., no encoding/decoding), we refer to this algorithm as *full-precision (parallel) SGD*. In this case, at each processor, if \mathbf{x}_t was the value of \mathbf{x} that the processors held before iteration t , then the updated value of \mathbf{x} by the end of this iteration is $\mathbf{x}_{t+1} = \mathbf{x}_t - (\eta_t/K) \sum_{\ell=1}^K \tilde{g}^\ell(\mathbf{x}_t)$, where each \tilde{g}^ℓ is a stochastic gradient. The speedup comes from the fact that all the updates are computed in parallel.

2.2 One-Bit Stochastic Gradient Descent

We describe the 1bitSGD quantization scheme, introduced by Seide et al. [39]. We denote the quantization function by $Q^{1b}(\mathbf{v})$, mapping a vector \mathbf{v} to its quantized version. The procedure first splits the vector \mathbf{v} into its positive and negative components, respectively, and computes the average over positive values, which we denote by avg_+ , and the average over negative values, which we denote by avg_- . Then,

$$Q_i^{1b}(\mathbf{v}) = \begin{cases} \text{avg}_+ & \text{if } v_i \geq 0 ; \\ \text{avg}_- & \text{otherwise.} \end{cases}$$

Simply put, the procedure replaces each component with the average corresponding to its respective sign. Critically, the procedure also maintains an error correction vector ϵ , associated to the model. In each iteration, the error from the previous iteration is *added* to the current value, before the value is quantized. This *error correction* step is critical to preserve accuracy [39]. The resulting algorithm works as Algorithm 2, which can be used as the Encode function of standard SGD (Algorithm 1).

2.3 SGD with Stochastic Quantization

We present the QSGD stochastic quantization scheme, following the description from the original paper [5]. We denote the quantization function by $Q_s(\mathbf{v})$, where $s \geq 1$ is a tuning parameter, corresponding to the number of “quantization levels”. Intuitively, s will define uniformly distributed levels between 0 and 1, to which each real value is quantized such that: 1) the value is preserved in expectation, and 2) minimal variance is introduced.

Given a non-zero vector $\mathbf{v} \in \mathcal{R}^n$, $Q_s(\mathbf{v})$ is defined as

$$Q_s(v_i) = \|\mathbf{v}\|_2 \cdot \text{sgn}v_i \cdot \xi_i(\mathbf{v}, s), \quad (1)$$

where $\xi_i(\mathbf{v}, s)$ ’s are independent random variables defined as follows. Let $0 \leq \ell < s$ be an integer such that $|v_i|/\|\mathbf{v}\|_2 \in [\ell/s, (\ell+1)/s]$. That is, $[\ell/s, (\ell+1)/s]$ is the quantization interval corresponding to $|v_i|/\|\mathbf{v}\|_2$. Then

$$\xi_i(\mathbf{v}, s) = \begin{cases} \ell/s & \text{with probability } 1 - p\left(\frac{|v_i|}{\|\mathbf{v}\|_2}, s\right); \\ (\ell+1)/s & \text{otherwise.} \end{cases}$$

Here, $p(a, s) = as - \ell$ for any $a \in [0, 1]$. If $\mathbf{v} = \mathbf{0}$, then we define $Q(\mathbf{v}, s) = \mathbf{0}$. The quantization distribution $\xi_i(\mathbf{v}, s)$ is defined to have minimal variance over distributions with support $\{0, 1/s, \dots, 1\}$. Its expectation satisfies $E[Q_s(v_i)] = v_i$. The quantizer is an unbiased estimator of the original gradient, thus ensures convergence [5, 14].

The above algorithm assumes quantization levels are uniformly distributed. There are algorithms in which quantization levels are distributed to further minimize variance, and they can, in some cases, significantly improve accuracy for model compression [50]. We implemented this for gradient but does not observe significant improvement.

2.4 Communication Primitives

The major communication bottleneck in the parallel SGD algorithm is in line 4 of Algorithm 1, where the gradient is broadcast to all other machines. Since the operation just needs to collectively add all gradients and produce the output at each node, this can be implemented using a standard instance of the `allreduce` operator, the optimized version of which has been studied for decades [32]. This operation can be implemented differently in standard CNTK: via a `reduce-and-broadcast` pattern implemented on top of MPI, which is the default in CNTK, and via NVIDIA’s NCCL extensions, which provide an `allreduce-sum` operation implementation.

2.4.1 MPI Reduce-and-Broadcast. The first aggregation algorithm is an MPI implementation of the classic `reduce-and-broadcast` pattern. More precisely, given a set of K nodes, the model of dimension n is split into n/K consecutive ranges, where the i th processor is logically assigned range $i[n/K]$. At the end of each processing batch, each processor has a full set of gradients, and the goal of the procedure is to aggregate (sum) all these gradients so that each processor has the same global gradient value at the end of the procedure.

The first step in the aggregation process is that a processor sends each range in its current gradient to the corresponding processor. Thus, each processor sums up the values for its assigned range locally. In the final step, each processor broadcasts its aggregated range to all other processors. Note that, in the actual implementation, the gradients may be transmitted in quantized form. (That is, we add quantization and un-quantization steps before communication, and after the message is received.) This does not affect the pattern.

2.4.2 NCCL Accumulation. NCCL [7] (pronounced “Nickel”) is an open-source communication library provided by NVIDIA, that is designed to provide efficient multi-GPU collective operations in a topology-aware way. In particular, NCCL provides a `sum` collective operation, which can be used to implement gradient aggregation. Internally, NCCL works by building a system-aware communication efficient ring topology, on top of which the collective operation is applied in a step-by-step manner. To minimize the memory impact of storing multiple copies, NCCL splits large data buffers into small slices (4-16KB), and uses efficient peer-to-peer access to push data between GPUs. NCCL currently supports several collective types (e.g., `all-gather`, `all-reduce`, `broadcast`) and a few binary operations (e.g., `sum`, `product`).

3 SYSTEM DESCRIPTION

We implemented our system on top of the Microsoft Cognitive Toolkit (CNTK) [3], version 2.0 beta 9. Our code is released both as open-source and as a docker instance at github.com/ZipML/CNTK. We first provide a brief technical overview of CNTK, and then focus on the additions brought by our artefact.

3.1 CNTK

The Microsoft Cognitive Toolkit (CNTK) is a computational platform optimized for deep learning. One general principle behind CNTK is that neural network operations are described by a directed computation graph, in which leaf nodes represent input values or network parameters, and internal nodes represent matrix operations on their children. CNTK supports and implements several popular network architecture types, such as feed-forward DNNs, convolutional nets (CNNs), and recurrent networks (RNNs/LSTMs). To train such networks, CNTK implements stochastic gradient descent (SGD) with automatic differentiation. CNTK supports parallelization across multiple GPUs and servers, with efficient MPI-based communication.

CNTK provides MPI-based GPU-to-GPU communication, and implements a CUDA-optimized version of the 1bit-SGD algorithm [39], as presented in Section 2.2. CNTK is optimized for usage with multi-server multi-GPU systems. CNTK exports APIs for C++, Python and C#/.NET. Additionally, CNTK specifies and implements a BrainScript scripting language, which can be used to define models such as neural networks, as well as to train and evaluate models without employing any lower-level programming languages. CNTK comes with many preinstalled scripts for training state-of-the-art models for different tasks, of which one of the most documented is image classification. Furthermore, it offers a number of pre-trained state-of-the-art models.

3.2 Low-Precision Support

CNTK implements 1bitSGD, which is used by default in multi-GPU environments with SGD. We now describe CNTK’s 1bitSGD implementation. Low-precision communication between GPUs is implemented using MPI `reduce-and-broadcast`.

3.2.1 Data Representation and 1bitSGD. CNTK stores the computation intensive data (model and minibatch samples) in the GPU memory, to ensure fast access and to avoid expensive data copying between host (main) and device (GPU) memory. In synchronized SGD, after every GPU is finished with the computation of gradients using backpropagation for its batch of samples, GPUs collectively aggregate gradients, so that each is left with a coherent copy of the model.

Gradients are stored using a matrix datatype in GPU memory. Aggregation is performed using the MPI Reduce-and-Broadcast technique described above. Sending gradient matrices is done separately for each gradient. To reduce communication, each GPU performs quantization and un-quantization on each message sent/received.

Quantization is done over columns. That is, the gradient is divided into columns, and each column is quantized separately. Thus, each quantized column is represented by two numbers avg_+ and avg_- , as well as an array of bits of size equal to number of elements in the column. That array of bits is represented as an integer array, where 8 quantized values, each consisting of 1 bit, are packed into 1 byte of memory. For instance, if there are n rows in a gradient matrix, then the quantized column is represented by two floating-point numbers and $\lceil n/8 \rceil$ bytes of memory.

In order for GPU memory to be aligned, if single-precision float-point numbers are used for calculations in a network, then avg_+ and avg_- are represented as a float data type, and n bits are packed in $\lceil n/32 \rceil$ C++ unsigned integers. If double-precision float-point numbers are used, avg_+ and avg_- are doubles and n bits are packed inside $\lceil n/64 \rceil$ C++ unsigned long long integers.

The quantization of a column is divided into 2 phases. The first phase launches GPU threads in order to calculate the numbers avg_+ and avg_- , where the number of threads is tuned for performance. The second phase uses calculated numbers avg_+ and avg_- to quantize values and pack bits into integers. For each integer in the quantized array, a new GPU thread is launched, which means that each thread in second phase quantizes exactly 32 or 64 elements of a column, depending on the precision employed. All of a gradient’s columns are quantized in parallel on the GPU. To optimize the performance, the *double buffering* technique is used, by which, while some gradients are being quantized, gradients that are finished with quantization are already being sent. That way communication overlaps with computation and GPU and CPU resources are maximally used. The current CNTK implementation uses MPI to exchange data between GPUs. Because of that, an additional transfer of the gradient between GPU device memory and host memory is required.

3.2.2 QSGD Implementation. In order to implement the QSGD quantization technique, we started from the 1bitSGD implementation. We re-wrote the quantization function, such that it follows the algorithm in Section 2.3. As in a case of 1bitSGD, we pack quantized values in integers, but we stored only one additional floating point number (for scaling) instead of two. The algorithm works such that the number of bits used for quantization is specified as $gBits$, and each gradient matrix values are quantize into an integer range with a specified number of bits. We implemented two different quantization methods, which differ in terms of the distribution of quantization levels. The first method follows faithfully the described algorithm: we use one out of $gBits$ bits to store the sign, and the rest of the bits is used for the quantized value. A second approach was to divide the interval $[-\|\mathbf{v}\|_2, \|\mathbf{v}\|_2]$ into $2^{gBits} - 1$ intervals of equal size, where the quantization levels are the endpoints of those intervals.

Since the dimensions of the gradient matrix could be large, and the variance introduced by quantization depends on the dimension, we implemented a variant which splits the vector into *buckets* of consecutive scalar values, where each bucket is quantized independently. The bucket size will be tuned for accuracy.

Dataset	# Training samples	# Validation samples	Size	# classes	Task
ImageNet	1.3M	50k	145GB	1000	Image
CIFAR-10	50k	10k	1GB	100	Image
AN4	948	130	64MB	NA	Speech

Figure 1: Statistics of datasets.

	Instance	# CPU cores	GPU	TFLOPS (single)	\$/hour
Amazon	p2.xlarge	4	1 × K80	1 × 8.73	\$0.9
	p2.8xlarge	32	8 × K80	8 × 8.73	\$7.2
	p2.16xlarge	64	16 × K80	16 × 8.73	\$14.4
DGX1		2 × 16	8 × P100	8 × 10.6	\$50 (Nimbix)

Figure 2: Statistics of machines

Importantly, the whole matrix is reshaped such that each column has a specified number of elements, where we always try to place consecutive elements from the same column in original matrix in the same bucket. Using this bucketing approach, we are able to control quantization variance and get significant accuracy improvements.

We also implemented possibility to specify how the scaling factor for a vector is done. Currently, we support normalization by 2-norm and maximal element of a vector (infinity norm). The former is useful if we wish to obtain sparse quantized vectors, while the latter introduces smaller variance. In our experiments, normalizing by the maximal element gave better results in terms of accuracy, since more information is preserved. We used the NVIDIA cuRAND library with a different seed for each thread to generate random numbers.

Additional improvement was not to quantize matrices with small number of elements compared to total number of learning parameters in the model, since for those matrices we lose time by setting GPU threads and quantizing. We use full precision pipeline to send those matrices. We choose a threshold for small matrices in such way so we always quantize more than 99% of all parameters.

Reshaped 1bitSGD. We noted that some of the design choices made in the CNTK implementation of 1BitSGD can adversely affect the performance of this algorithm. For objects without dynamic dimensions, the first tensor dimension is the “row,” while the other dimensions are flattened onto “columns.” The CNTK implementation of 1BitSGD always quantizes *per column*.

Practically, on networks with many convolutional layers, this can lead to the following performance artefact: quantization is often applied to a column of very small dimension (1–3), which is computationally expensive, and leads to practically no communication reduction. (At the same time, this artefact leads to practically no accuracy loss for this version of the algorithm, since the gradients are almost unchanged.) Given this artefact, the standard implementation of 1BitSGD can be slower than even the full-precision version on heavily convolutional networks such as ResNet and Inception.

We correct this issue by always reshaping tensors with a reshaping technique applied in QSGD. We observe up to 4× speedup compared with the original CNTK implementation. In all experiments we will denote this version of 1bitSGD with 1bitSGD*.

4 EXPERIMENTAL SETUP

4.1 DataSets

We test various quantization techniques on two types of tasks: image classification, and automated speech recognition.

Task	Network	Dataset	Params.	# epochs to run	Initial Learning Rate
Image	AlexNet	ImageNet	62M	112	0.07
	BN-Inception	ImageNet	11M	300	3.6
	ResNet50	ImageNet	25M	120	1
	ResNet110	CIFAR-10	1M	160	0.1
	ResNet152	ImageNet	60M	120	1
	VGG19	ImageNet	143M	80	0.1
Speech	LSTM	AN4	13M	20	0.5

Figure 3: Statistics of networks.

	1 GPU	2 GPUs	4 GPUs	8 GPUs	16 GPUs
AlexNet	256	256	256	256	256
BN-Inception	64	128	256	256	256
VGG19	32	64	128	128	128
ResNet50	32	64	128	256	256
ResNet152	16	32	64	128	256
ResNet110	128	128	128	128	128
LSTM	16	16	NA	NA	NA

Figure 4: Batch sizes used for each network and # GPUs. See Section 4.4 for the tuning protocol that results in these choices.

ImageNet. The ILSRVC 2015 dataset [37] (ImageNet) consists of 1.3 million images, each labelled into one of 1000 categories (classes), which cover a wide variety of objects, animals, scenes, or geometric concepts. To evaluate our models, we consider a standard scenario where validation set that consists of 100K images is used. The classifier has to predict either a single label (top-1) or five labels (top-5); an image is considered to be correct if at least one of the outputs matches the ground truth.

CIFAR-10. We also consider the small-scale CIFAR-10 object classification dataset [26]. The training set consists of 50,000 32×32 images. Images are labelled into 10 categories. The classifier has to output a single label (top-1); an image is considered to be correct if the prediction matches the ground truth.

AN4. For speech recognition, we use the CMU Alphanumeric (AN4) dataset [2]. AN4 consists of recordings of subjects spelling out their census data (name, address), as well as randomly generated sequences. It contains 948 training utterances, and 130 test utterances, sampled at 16 kHz, with 16-bit linear sampling.

4.2 Networks

We conducted our experiments on state-of-the-art networks for image classification and speech recognition.

For image classification we tested all the winning entries from the ImageNet competition from 2012 to 2015: AlexNet, VGG, BN-Inception, and ResNet. These architectures are state-of-the-art for this task. For experiments with VGG we used an instance with 19 layers, called VGG19. We performed experiments on the ImageNet dataset using ResNet networks with 50 and 152 layers (called ResNet50 and ResNet152), and used an instance of ResNet110 for the CIFAR10 dataset. All of these networks have different properties and different types of layers. BN-Inception is optimized for low number of parameters in network, whereas ResNet is built entirely of convolutional layers. For automated speech recognition we used a network that consists of 3 long short-term memory (LSTM) components. In all experiments we used architectures built and optimized by CNTK. More details about each network are in Figure 3.

4.3 Machines

For experiments, we used Amazon AWS EC2 P2 instances, designed for high-performance computing using GPUs. Instances we used are p2.xlarge with 1 GPU, p2.x8large with 8 GPUs on a single machine and p2.x16large with 16 GPUs on a single machine. The above instances have Nvidia Tesla K80 GPUs, based on the Kepler architecture. These GPUs support GPUDirect, which enables peer-to-peer GPU communication without using CPU.

Another machine we used is DGX1 that consists of 8×P100 GPUs. These GPUs are based on Pascal architecture and have NVlink high-bandwidth interconnection between GPUs. This results in highly-optimized GPU communication. The DGX machine also benefits from a customized interconnect, which provides high throughput and low latency.

4.4 Tuning and Experimental Protocol

We used nvidia-docker to conduct our experiments. One complicating factor of benchmarking deep learning systems is a large set of hyperparameters one needs to tune to conduct fair comparison. In this paper, we use the following tuning protocol.

Batch Size. The principle behind our hyperparameters tuning protocol is to always start from CNTK’s default hyperparameters, which have already been optimized by the CNTK developers for a specific network. For each configuration that we run, if CNTK does not run with the default hyperparameter setting, we vary the hyperparameter until it runs. For example, with the default batch size and 1GPU, CNTK cannot run ResNet because the whole batch does not fit in memory. In this case, we decrease the batch size until CNTK runs. Figure 4 shows the batch sizes we used.

Learning Rate. We always use the default learning rate, which is fine tuned by CNTK for 32-bit full precision. We use an SGD optimizer with default momentum (0.9 for most architectures). Note that, because we focus on synchronous communication, the number of GPUs has negligible impact on the convergence rate and final quality given a fixed value of the learning rate and momentum. This has been observed previously, in e.g. [24].

Bucket Size. QSGD and 1bitSGD with reshaped matrices introduce new hyperparameters, such as bucket size and normalization scaling factor. We picked these parameters that optimize for accuracy on these networks. We used QSGD 2bit with bucket size 128, QSGD 4bit and 8bit with bucket sizes 512, and QSGD 16bit with bucket size 8192. Also, for our version of 1bitSGD with reshaped matrices we tried multiple bucket sizes and used bucket size 64 for all performance and scalability numbers.

NCCL Simulation. CNTK uses NCCL only for full-precision data parallel SGD. When trying to twist it for low-precision, we find that its sum primitive only supports full precision and therefore, cannot be used for low-precision communication (as scaling factors cannot be taken into account). Thus, we implement a version where we simulated gradient aggregation with NCCL calls—we send the same number of bits using NCCL as we would if NCCL would have had support for low-precision. In the NCCL experiments, we use this simulated version to compare the performance of different low-precision setups.

5 ANALYSIS

We now analyze the experiment results and discuss the insights.

5.1 Accuracy

Does low-precision always hurt accuracy?

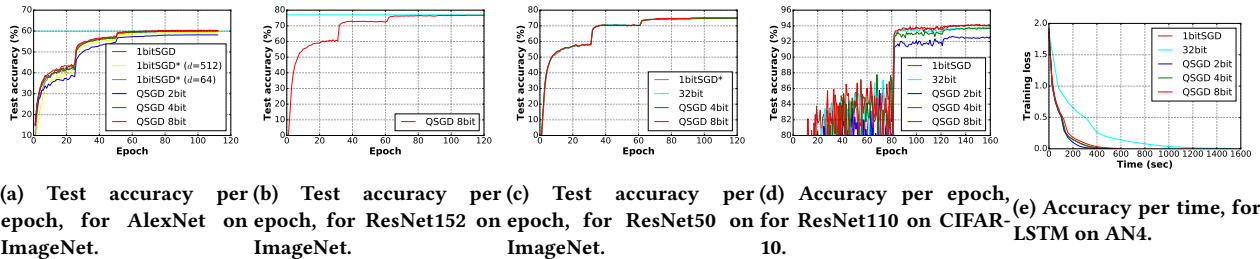


Figure 5: Accuracy results for various networks and datasets. Cyan = 32bit.

In the first set of experiments, we study the impact of training deep neural networks with low precision communication on the training accuracy. For each network and data set, we vary the precision and report the test accuracy¹ for each epoch. Figure 5 illustrates the result for AlexNet, ResNet50 and ResNet152 on ImageNet, ResNet110 on CIFAR, and an LSTM network on AN4.

Accuracy: Positives. There always exist settings of parameters for which quantized variants converge to the same or better final accuracy, compared to the full-precision version. For ResNet50/ImageNet, the full precision variant converges to 59.90% top-5 accuracy, whereas 1bitSGD converges to 60.31% accuracy. Similarly, QSGD with 4bit and 8bit quantization converge to 60.37% and 60.05% final accuracy, respectively. The accuracy improvement, of up to 0.47% is statistically significant. The reason for this small, but statistically significant, accuracy gain is not clear. We suspect it is caused by similar reasons with recent work observing that adding noise during the training process can improve accuracy [35]. Additional experiments show that, for this dataset, quantization also improves *training loss* (see Figure 5e), which is perhaps surprising. It is not clear what is the definitive reason of this behavior.

Accuracy: Negatives. It is also important to note that quantizing too aggressively can lead to significant accuracy loss. For example, QSGD where gradient values are quantized to two bits (levels 0, 1, and -1) has accuracy loss of at least 1 percentage point, consistently across image classification datasets. We note that this is not the case for non-convolutional networks (LSTMs), which appear to be able to handle quantization to very low precision. This finding is consistent with the theory [5, 50]: aggressive quantization increases the variance of the convergence process, and as such, the theory predicts that one would have to run for more iterations in order to converge to the same quality results.

Convergence Rate. Another aspect of accuracy is the *convergence rate*, i.e., how many epoches do the system need to converge to the same accuracy. We observe that 8bit bit QSGD with 512 bucket size is always enough, on all data sets considered, to guarantee effectively the same convergence rate. For 1bitSGD, we note that the reshaped variant requires relatively small bucket size (64) to converge to the same target accuracy. Even so, we note that the convergence rate of 1bitSGD (i.e., training error versus epoch) is lower per epoch compared to that of the full precision variant, but the best accuracy achieved across at the end of the training process is comparable for the two processes.

Impact of Layer Types. We also find *convolutional layers* are more “sensitive” to the noise induced by quantization. This phenomenon becomes apparent when we study the accuracy of two variants: (1) only quantize convolutional layers, and (2) quantize all layers. For example, on AlexNet, we can compare the accuracy of 1bitSGD with reshaping, which quantizes all layers, with the accuracy of standard 1bitSGD, which effectively does not quantize convolutional layers, as previously discussed. We observe that the reshaped variant has end accuracy that is lower; a closer examination of the loss-per-epoch graph shows that the reshaped version has consistently lower accuracy throughout the training process, although it almost catches up in terms of final accuracy across all epochs. The accuracy difference is starker (1 accuracy point) if we examine reshaped 1bitSGD with 512 bucket size.

Impact of Bucket Size. Another factor that has impact on the training accuracy is the bucket size of quantization, an additional parameter which we implemented for reshaped 1bitSGD and QSGD, and tuned for accuracy. For QSGD, this parameter can be used to directly throttle the added variance of the quantization process, at the cost of extra communication (since we need to send an extra floating-point number per each bucket). We observed that this can make a significant difference in terms of accuracy. For instance, on AlexNet / ImageNet, 4bit QSGD with 8192 bucket size has end accuracy that is $> 0.6\%$ inferior to the full-precision variant. Adjusting the bucket size to 512 allows it to *improve* accuracy over the full precision variant.

Further Lowering Accuracy. We also experimented with variants of QSGD with even lower communication overhead (such as 1-bit per location, or two-norm normalization). These variants are theoretically justified, in that they will eventually converge to a local optimum, at the cost of additional running time. However, in our experiments, these variants did not provide good accuracy results when run for the same number of epochs as the full precision version. These accuracy results are therefore omitted.

Discussion. One final observation we make is the impressive accuracy of the 1bitSGD error-correction techniques. It is worth emphasizing that this technique only sends the *signs* of the components, plus two scaling factors. On large-scale image classification datasets, it only loses relatively small amounts of accuracy ($< 0.3\%$). It is interesting future work to develop the theoretical foundation of its convergence and correctness, which is still an open question [39].

5.2 Performance

Does low-precision always help performance?

We now study the impact of low-precision training on end-to-end training time. We measure the time per iteration for the

¹We also conduct similar experiments for training accuracy. We leave the result to the full version as it does not change any claims.

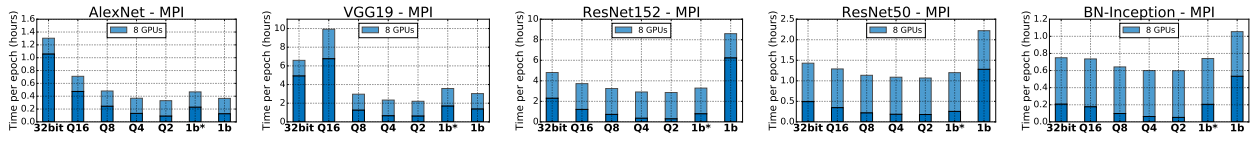


Figure 6: Performance: Amazon EC2 Instance with MPI. QN = QSGD with N bits. 1b = 1bitSGD, 1b* = 1bitSGD with reshaping.

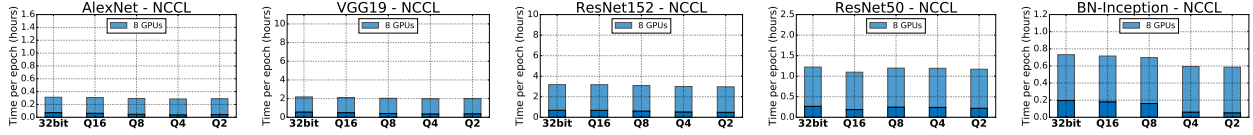


Figure 7: Performance: Amazon EC2 Instance with NCCL. QN = QSGD with N bits. 1b = 1bitSGD, 1b* = 1bitSGD with reshaping.

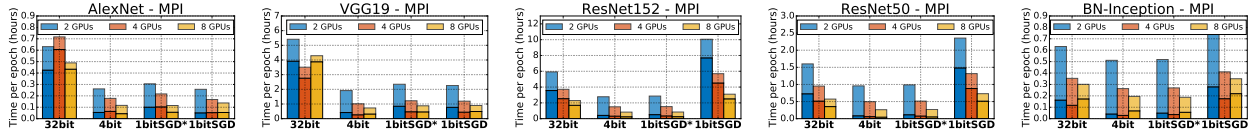


Figure 8: Performance: NVIDIA DGX-1 with MPI.

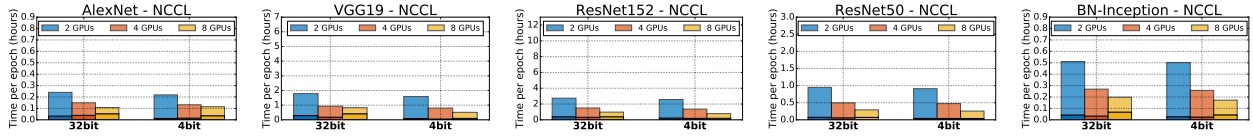


Figure 9: Performance: NVIDIA DGX-1 with NCCL.

full-precision version (32bit), the standard CNTK implementation of 1bitSGD, our variant with reshaping and 64 bucket size (whose parameters are chosen so as to guarantee no accuracy loss), and QSGD with 2, 4 and 8-bit precision. (As observed, 512 bucket size guarantees state-of-the-art accuracy across all networks for the QSGD 4bit and 8bit variants. Even lowering the bucket size down to 32 for 2bit QSGD does not recover the full precision accuracy.) We break down the epoch time into communication time (bottom of each bar), and computation time (top of each bar, which also includes time spent compressing and uncompressing gradients). Figure 6 and Figure 7 show the result on the 16GPU Amazon instance, and Figure 8 and Figure 9 show the results on the 8GPU NVIDIA DGX-1. (Recall that DGX-1 machine has newer Pascal GPUs, as well as a faster, custom interconnect.)

Slow Inter-connections with Slow Primitives. We start with a setting that favors low precision communication the most — Amazon instances have 16 GPUs and all communications are conducted over MPI/PCIe. Figure 6 illustrates the result of using MPI as the communication primitives. We see that, in this setting, using low-precision communication significantly improves the performance — with 8GPUs on VGG network, the speedup of using 2-bit / 4-bit precision is almost 3× compared with using 32-bit full precision. On 16GPUs, the speedup is of > 5×.

This speedup does not hold across all network types, as these networks have different ratio between communication and computation: there are communication-dominated networks (such as

AlexNet and VGG), and computation-dominated networks (such as BN-Inception and ResNet50). ResNet152 balances these two.

Impact on Communication Overhead. We observe that on all networks, lowering the precision of communication significantly decreases the communication time. On AlexNet, the reduction in communication time with 4-bit quantization is almost 5×. For other networks, we observe similar speedups.

Impact on End-to-end Performance. As the computation time stays the same across different precision settings, the end-to-end performance speedup is smaller than our amount of savings we have in communication. For example, on AlexNet, the 5× speedup on communication results in 2× speedup overall. For computation-dominating network such as BN-Inception, we get only 1.3× speedup when lowering the precision by 16×.

Extremely Low Precision. This “diminishing returns” phenomenon implies that, even on machines with slow connection and slow communication, we rarely observe a case where using 1-2bit precision resulting in significant performance benefit as using 8-bits and 4-bits. In practice, this suggests that it may be reasonable to only lower the precision up to 8-bit gradients, since that going even lower does not really provide significant performance benefit on the model sizes we trained.

Precision versus Bucket Size. We observe that in many cases 1bitSGD is actually slower than 2bit QSGD. This is because, to preserve accuracy, we have to significantly decrease bucket size

AlexNet ImageNet						
Setup		Samples per second (MPI)				
Precision	Bucket size	1 GPU	2 GPUs	4 GPUs	8 GPUs	16 GPUs
32bit	/	240.80	301.45	328.00	272.90	192.10
QSGD 16bit	8192	/	388.80	508.80	500.90	335.60
QSGD 8bit	512	/	424.90	544.60	739.10	535.00
QSGD 4bit	512	/	466.50	598.70	964.90	748.50
QSGD 2bit	128	/	449.20	609.15	1076.50	889.80
1bitSGD	/	/	424.05	564.30	971.10	849.40
1bitSGD*	64	/	370.80	476.50	761.20	712.70
1bitSGD*	512	/	/	/	/	/

ResNet50 ImageNet						
Setup		Samples per second (MPI)				
Precision	Bucket size	1 GPU	2 GPUs	4 GPUs	8 GPUs	16 GPUs
32bit	/	47.20	80.80	142.40	247.90	272.30
QSGD 16bit	8192	/	90.20	156.30	275.80	348.70
QSGD 8bit	512	/	92.60	162.70	313.70	416.80
QSGD 4bit	512	/	93.90	165.70	326.10	461.20
QSGD 2bit	128	/	93.30	178.35	330.45	472.25
1bitSGD	/	/	45.10	81.70	160.15	155.20
1bitSGD*	64	/	88.10	156.50	296.70	442.40

ResNet110 CIFAR10						
Setup		Samples per second (MPI)				
Precision	Bucket size	1 GPU	2 GPUs	4 GPUs	8 GPUs	16 GPUs
32bit	/	343.70	555.00	957.70	1229.10	831.60
QSGD 16bit	8192	/	551.00	942.70	1164.20	763.40
QSGD 8bit	512	/	550.20	960.10	1193.10	759.70
QSGD 4bit	512	/	571.10	957.40	1257.10	784.30
QSGD 2bit	128	/	557.20	973.10	1227.90	780.40
1bitSGD	/	/	465.60	643.30	610.90	406.90
1bitSGD*	64	/	550.40	884.80	1156.70	757.70

ResNet152 ImageNet						
Setup		Samples per second (MPI)				
Precision	Bucket size	1 GPU	2 GPUs	4 GPUs	8 GPUs	16 GPUs
32bit	/	16.90	26.10	45.00	73.90	113.50
QSGD 16bit	8192	/	31.20	54.50	95.50	151.00
QSGD 8bit	512	/	32.80	62.70	109.20	182.50
QSGD 4bit	512	/	33.60	60.20	121.90	203.20
QSGD 2bit	128	/	33.50	64.35	123.55	208.50
1bitSGD	/	/	10.55	22.10	41.40	63.15
1bitSGD*	64	/	30.40	55.50	108.10	193.50

VGG19 ImageNet						
Setup		Samples per second (MPI)				
Precision	Bucket size	1 GPU	2 GPUs	4 GPUs	8 GPUs	16 GPUs
32bit	/	12.40	20.40	36.30	53.95	40.60
QSGD 16bit	8192	/	24.80	46.40	35.80	67.80
QSGD 8bit	512	/	24.20	47.50	119.50	106.60
QSGD 4bit	512	/	27.00	52.30	151.65	143.80
QSGD 2bit	128	/	24.60	49.35	160.35	170.50
1bitSGD	/	/	22.20	43.15	117.35	120.60
1bitSGD*	64	/	22.90	44.80	99.15	134.30

BN-Inception ImageNet						
Setup		Samples per second (MPI)				
Precision	Bucket size	1 GPU	2 GPUs	4 GPUs	8 GPUs	16 GPUs
32bit	/	88.30	164.80	316.75	473.75	500.40
QSGD 16bit	8192	/	171.80	337.10	482.70	592.30
QSGD 8bit	512	/	173.60	342.50	552.90	696.30
QSGD 4bit	512	/	174.80	346.90	593.40	743.30
QSGD 2bit	128	/	173.40	343.70	591.80	747.50
1bitSGD	/	/	127.60	236.25	336.15	321.30
1bitSGD*	64	/	170.30	335.10	480.50	700.40

Figure 10: Speed of using MPI on Amazon EC2 instance.

for the reshaped 1bitSGD algorithm. This renders it both more expensive in terms of communication, and increases the computational complexity of the GPU encoding/decoding operations.

Slow Inter-connections with Fast Primitives. Another way to implement the communication is to use NCCL instead of MPI. In our experiment, for the primitives we used in training, NCCL provides significantly faster communication. The reasons are two-fold: NCCL provides faster messaging, but also less memory overhead. Figure 7 shows the result of using NCCL on EC2.

Implementation Notes. We note the following caveats when using NCCL. First, NCCL does not currently support more than 8 GPUs, and we therefore only report numbers on up to 8 GPUs. Second, the current allreduce primitive in NCCL does not support low-precision and therefore all low-precision numbers of

AlexNet ImageNet					
Setup		Samples per second (NCCL)			
Precision	Bucket size	1 GPU	2 GPUs	4 GPUs	8 GPUs
32bit	/	240.80	458.20	625.00	1138.30
QSGD 16bit	8192	/	462.80	632.10	1157.60
QSGD 8bit	512	/	458.40	641.80	1214.80
QSGD 4bit	512	/	471.90	659.40	1247.70
QSGD 2bit	128	/	471.00	661.60	1229.70

ResNet50 ImageNet					
Setup		Samples per second (NCCL)			
Precision	Bucket size	1 GPU	2 GPUs	4 GPUs	8 GPUs
32bit	/	47.20	93.80	164.80	291.10
QSGD 16bit	8192	/	93.70	164.50	324.20
QSGD 8bit	512	/	94.00	165.80	297.40
QSGD 4bit	512	/	95.60	167.90	298.40
QSGD 2bit	128	/	95.50	168.20	304.10

ResNet152 ImageNet					
Setup		Samples per second (NCCL)			
Precision	Bucket size	1 GPU	2 GPUs	4 GPUs	8 GPUs
32bit	/	16.90	33.60	60.10	112.10
QSGD 16bit	8192	/	33.40	59.80	112.20
QSGD 8bit	512	/	33.70	60.80	115.10
QSGD 4bit	512	/	34.20	62.10	118.70
QSGD 2bit	128	/	34.30	62.20	119.90

VGG19 ImageNet					
Setup		Samples per second (NCCL)			
Precision	Bucket size	1 GPU	2 GPUs	4 GPUs	8 GPUs
32bit	/	12.40	24.90	48.70	163.10
QSGD 16bit	8192	/	24.90	49.10	168.00
QSGD 8bit	512	/	25.50	50.50	175.20
QSGD 4bit	512	/	25.60	51.00	179.50
QSGD 2bit	128	/	25.60	51.10	177.80

BN-Inception ImageNet					
Setup		Samples per second (NCCL)			
Precision	Bucket size	1 GPU	2 GPUs	4 GPUs	8 GPUs
32bit	/	88.30	173.30	342.00	486.70
QSGD 16bit	8192	/	174.30	342.70	497.10
QSGD 8bit	512	/	174.50	345.30	510.10
QSGD 4bit	512	/	178.60	349.00	598.90
QSGD 2bit	128	/	177.20	349.00	608.20

Figure 11: Speed of using NCCL on Amazon EC2 instance.

Figure 7 are simulated by using the NCCL allreduce primitive to send the same amount of data that would be sent in a low-precision implementation using NCCL. (The rest of the algorithm remains the same; in this case, the GPUs will converge at a lower rate or could diverge, but this is irrelevant for the experiment.)

NCCL vs. MPI. We draw the reader’s attention to the performance difference between MPI (Figure 6) and NCCL (Figure 7). It is clear that the MPI implementation is slower than the NCCL – the computation time stays the same but the communication time differs significantly. One result we found especially surprising is that NCCL with full precision can be faster than MPI with low precision. Thus, on systems where NCCL is available, the fastest approach may be simply to run full precision with NCCL.

Super-Linear Scaling. The attentive reader may have noticed that, for the VGG19 network, the scaling is super-linear at 8GPUs. This phenomenon appears to be an artifact due to the minibatch size: at 8 GPUs, the batch for VGG19 consists of just 16 images, which the K80 GPUs are able to process in less than half of the time needed to process batches of 32 images. We were able to reproduce this behaviour on a single GPU with the same batch size, and we speculate that it may be due to better caching and less data movement at smaller sample size.

End-to-end Performance. Since NCCL could be modified to supports a low-precision allreduce in the future, we might be able to take advantage of low-precision communication to get even faster systems. As illustrated in Figure 7, on communication-dominated networks such as VGG, we get up to 1.5× speedup with 4-bit or 8-bit precision NCCL. This end-to-end speedup is

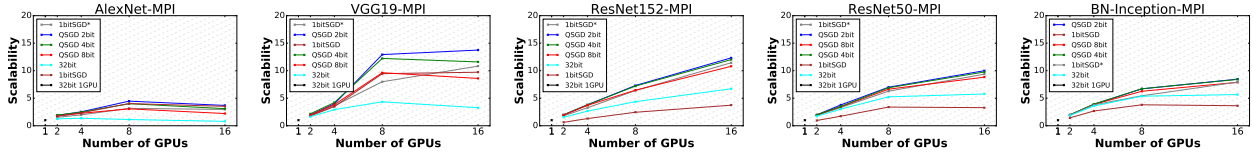


Figure 12: Scalability: Amazon EC2 Instance with MPI.

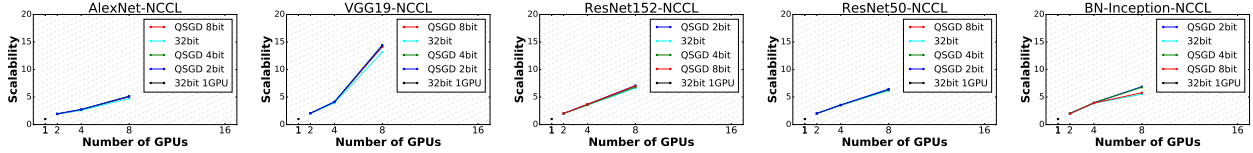


Figure 13: Scalability: Amazon EC2 Instance with NCCL.

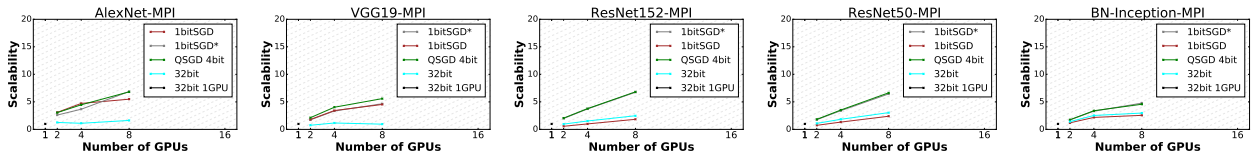


Figure 14: Scalability: NVIDIA DGX-1 with MPI.

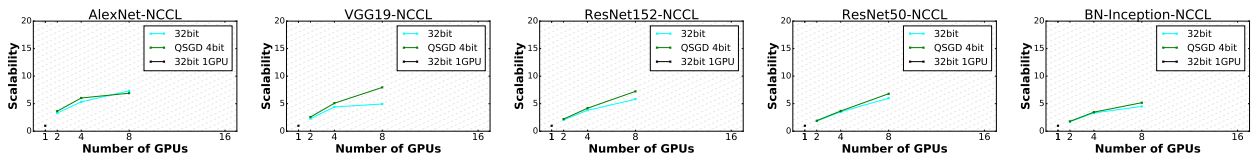


Figure 15: Scalability: NVIDIA DGX-1 with NCCL.

considerably lower than what we got in the MPI implementation, due to the balanced communication-to-computation ratio.

Fast Interconnect with Slow/Fast Primitives. On NVIDIA DGX-1, all GPUs are connected by a custom interconnect, which is significantly faster than PCIe. Further, the GPU is about 40% faster than in the Amazon instances. In this case, we observe similar results as the Amazon instances – when using slow primitives such as MPI, we get significant speedup by lowering the precision of communication, sometimes by up to 5× (VGG). However, with NCCL, the achievable speedup is again limited significantly. In fact, with NCCL, on VGG, we get up only 1.6× speedup when using 4-bits precision and relatively minor speedups for other networks and precision levels. It is interesting to note by examining the scalability graphs that the low precision brings the implementations close to linear scalability.

5.3 Scalability

Do we really need 16 GPUs on a single instance?

We now study scalability: how does the performance change with respect to the number GPUs we use? We define scalability as the number of samples per second in a certain configuration, divided by the number of samples per second processed by a single GPU for a training epoch. Figure 12-15 shows the result.

Scalability of Full Precision. We see that, using 32-bit full precision is able to scale, to some extent. For computation-dominated networks, 32-bit full precision is able to scale up to 6× with MPI and 7× with NCCL on 8 GPUs. However, for communication dominated networks, 32-bit suffers. For example, for AlexNet, 32-bit full precision with MPI only achieves 2× scale up with 16 GPUs. Comparing NCCL with MPI, NCCL scales up better than MPI for 32-bit full precision – This is not surprising, as NCCL is more efficient in dealing with communications.

Impact of Low Precision Communication. We see that using low-precision quantized communication consistently improves the scalability over 32-bit for all experiments. When using MPI on Amazon instances, the scalability for AlexNet is improved from < 3× to 8× by using 1bitSGD. Networks such as ResNet152 scale almost *linearly* once quantization is applied even with MPI: with quantization, scalability at 16GPUs is 2×, and 5× without.

When using NCCL, the difference between quantized communication and full precision decreases significantly. As shown in Figure 13 and Figure 15, quantization only introduces at most 50% scale up compared with full precision. The only exception is for VGG, which has the most number of parameters, in which 32-bit saturated the memory bandwidth. On the other hand, on AlexNet, we note that 32bit NCCL is *faster* than the 4bit variant. This explanation for this perhaps surprising result is the following: the computational overhead of AlexNet is relatively low; 4bit quantization adds to it, since we need to compress and

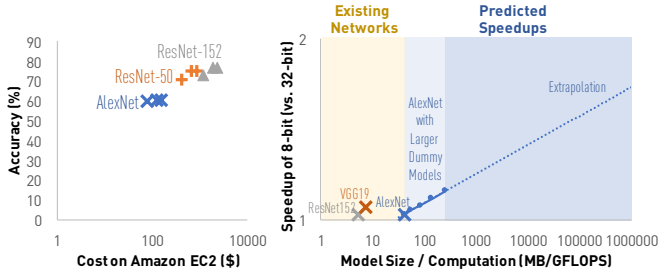


Figure 16: Left: Price and accuracy of training different neural networks with different # epochs on Amazon EC2 with 8-bit and NCCL. Right: The performance improvement of using low-precision communication w.r.t. the communication and computation ratio.

de-compress the data. This overhead is not compensated by the reduction in communication for this network.

Impact on Reshaping. One optimization we did to improve OneBitSGD is reshaping. The scalability of the reshaped version of 1bitSGD is initially lower than the original on e.g. AlexNet, due to the higher computational cost of quantizing into buckets, but this cost is amortized at higher thread counts. We note however that overall the less computationally expensive QSGD variant may perform better, although it sends more data per iteration.

5.4 Discussion

Accuracy vs. Cost. The experimental results we get in this paper also allow us to understand a tradeoff between the model accuracy and the dollar cost of training. Such tradeoff could be useful for scenarios like the following: assume a user with a large dataset (e.g., ImageNet), wishing to train a large-scale neural network to full precision, but on a limited budget. What is the most cost-effective way to do so, assuming current Amazon pricing?

Figure 16 illustrates the results of the price and accuracy for training different networks to full accuracy, according to its published recipe (accuracy and # of epochs to convergence). We use the cheapest EC2 solution for each network, which we derive from the scalability graphs. We see that there is an almost monotonic correlation between \$ cost and accuracy – the more \$ the user spent, the higher the quality s/he can expect. On the other hand, it is also clear that there exists a diminishing return phenomenon – spending \$600 to switch from AlexNet to ResNet-50 bumps the accuracy by 15 percentage points, but another \$1500 to switch to ResNet-152 only gets 2 percentage points improvement. It would be interesting to study an automatic management system that is both budget-aware and error tolerance-aware.

MPI vs. NCCL. The answer to this question may be now obvious to the reader. While low-precision techniques can render the MPI implementation competitive, the gap in performance is clear, since the NCCL implementation is heavily optimized. One issue is that NCCL is currently not fully supported for large GPU deployments, such as multi-node or supercomputer setups. In these cases, an MPI-based implementation is necessary. An interesting topic for future work would be to add or improve MPI support for such reduction operations, as well as support for low-precision data representations.

1bitSGD vs QSGD. Given our experiments, there is no clear winner between the two quantized methods. However, it is worth noting that the 8bit QSGD variant provides stable accuracy and

performance for all the architectures we consider, and therefore may be a good entry-level compressor. Further, we note that QSGD wins in most of the head-to-head comparisons, albeit by small margins. The 1bitSGD algorithm communicates the least data of all the methods we tried, and can provide impressive accuracy results. Thus, in a setting where tuning is possible or minimal communication is necessary, 1bitSGD may be a good alternative. Unfortunately, there is no easy way of predicting whether a quantized method will achieve the target accuracy for a network without actually running it.

6 OUTLOOK

One general way to interpret our results is that the neural network training workloads we considered are currently *not communication-bottlenecked*. More precisely, either by using compressed communication, NCCL, or both, all the networks we considered can achieve significant speedup on multiple GPUs.

What would happen to scalability if we extrapolate the trend of increasing the model size? In particular, *in what model-size-to-GFLOPS regime would the scalability impact of quantization be significant?* These are the questions we address in Figure 16, where we examine the performance improvement (in terms of speedup) of the 8-bit quantized variant over the full-precision one, as we (artificially) increase model size for the AlexNet architecture, for the 8GPU NCCL version, on which we only noted minimal improvements using quantization. The experiment shows that, the performance improvement due to low-precision communication depends on a key factor, i.e., the ratio between communication and computation (MB/GFLOPS). As the MB/GFLOPS ratio increases, running time becomes driven by the cost of communication, and hence the speedup because of quantization is more apparent. Notice that the overall speedup will always be upper bounded by the difference in bandwidth usage, which is $4\times$.

Another trend is the increased computational power of GPUs, and the presence of increasingly many GPUs on a single machine. The first factor decreases the computational cost of networks, while the second increases the overall communication bandwidth of the workload. We can therefore speculate that these factors will bring quantized methods closer to the fore in terms of ways to reduce total running time, although it is also likely that the speed of communication between GPUs will improve in the future.

7 RELATED WORK

Changing the precision of data representation in machine learning systems has received tremendous interest recently [5, 13, 14, 16–18, 20, 23, 28, 29, 33, 36, 39, 45, 50, 51]. These works cover a whole spectrum of machine learning tasks, from training to inference, as well as models, from deep learning to linear models. This paper builds upon this prior art, but focuses on the system perspective, to understand to what extent these approaches help.

Low-Precision Deep Learning Training The research area closest to our work is *training* deep learning models with low precision [3, 5, 18, 23, 39]. Most algorithms use *quantizing* different data channels to lower the precision of the data representation.

One of the first references to recognize the performance impact of gradient communication when training large speech models was 1BitSGD [39]. This algorithm is inspired by delta-sigma modulation [38] for analog-to-digital conversion. In further work, variants of the same algorithm are benchmarked and refined on large-scale Amazon proprietary datasets by Strom [42]. The

1BitSGD algorithm is available by default in the Microsoft Cognitive Toolkit (CNTK) [3]. Another algorithm we consider is QSGD [5], based on the idea of *stochastically rounding* floating-point values to a small set of integer levels. For both algorithms, it is not clear how well they generalize to other deep learning models and what are their system tradeoffs; if fact, 1BitSGD is even observed to diverge in some large-scale experiments by the original authors. The goal of this paper is to provide such a fair and well-optimized system benchmark.

Recent work considered alternative quantization strategies. Aji and Heafield [4] proposed to *truncate* the gradients to only the top few percentage of components—sorted by magnitude—and to store the remaining components locally, in an accumulation vector. This enables *sparse* communication to be employed, and the authors show that extremely small densities (<0.5%) are sufficient for convergence for neural machine translation tasks. This scheme is promising, and can be theoretically justified by relating it to *asynchronous* SGD. However, we believe that this method requires further research to be widely applicable, for the following two reasons. First, in ImageNet experiments on the Inception architecture, we noticed that the density levels required for convergence to the same accuracy level as the full communication variant were large (>10%); due to the extra cost of transmitting indices, it is not clear that the reduction in communication is sufficient to ensure scalability on such tasks. Second, sparse communication is not efficiently supported by communication primitives such as NCCL or MPI.

Another approach to reduced communication is to *factorize* large network layers, and communicate the factors instead of large matrices [11, 47]. These methods are effective at reducing communication for fully-connected layers, but are less useful for conv layers, where weights are typically smaller than activations. As many modern architectures, e.g. ResNet, are almost entirely convolutional, this could limit the usefulness of this approach.

Distributed Machine Learning There is certainly no shortage of distributed systems for machine learning, and deep learning in particular, such as TensorFlow [1], CNTK [3], Theano [44], Torch [12], Caffe [25], and MXNet [9]. The database community has also been contributing to this list intensively, and examples include MLib [31] built upon Spark [48], KeystoneML [41], SystemML [8], and GraphLab [30]. However, only a few of these systems support low precision training. We hope our study helps clarify the system tradeoff introduced by low precision communication and that the insights can help to improve these systems.

(Acknowledgment) DA is supported in part by the SNF Ambizione Fellowship. CZ and the DS3Lab gratefully acknowledge the support from the Swiss National Science Foundation NRP 75 407540_167266, IBM Zurich, Mercedes-Benz Research & Development North America, Oracle Labs, Swisscom, Zurich Insurance, Chinese Scholarship Council, and the Department of Computer Science at ETH Zurich, the GPU donation from NVIDIA Corporation, the cloud computation resources from Microsoft Azure for Research award program.

REFERENCES

- [1] M. Abadi et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *ArXiv*, 2016.
- [2] A. Acero. *Acoustical and environmental robustness in automatic speech recognition*, volume 201. Springer Science & Business Media, 2012.
- [3] A. Agarwal et al. An introduction to computational networks and the computational network toolkit. Technical report, Tech. Rep. MSR, 2014.
- [4] A. F. Aji et al. Sparse communication for distributed gradient descent. In *EMNLP*, 2017.
- [5] D. Alistarh, J. Li, R. Tomioka, and M. Vojnovic. QSGD: Randomized Quantization for Communication-Optimal Stochastic Gradient Descent. *NIPS*, 2017.

- [6] M. M. Astrahan et al. System R: relational approach to database management. *TODS*, 1976.
- [7] A. A. Awan et al. Efficient large message broadcast using NCCL and CUDA-aware MPI for deep learning. In *EuroMPI*, 2016.
- [8] M. Boehm et al. SystemML. *PVLDB*, 9(13):1425–1436, 2016.
- [9] T. Chen et al. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *Arxiv*, 2015.
- [10] S. Chetlur et al. cuDNN: Efficient Primitives for Deep Learning. 2014.
- [11] T. Chilimbi et al. Project adam: Building an efficient and scalable deep learning training system. In *OSDI*, 2014.
- [12] R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A Matlab-like Environment for Machine Learning. In *BigLearn, NIPS Workshop*, 2011.
- [13] C. Cortes, M. Mohri, and A. Talwalkar. On the Impact of Kernel Approximation on Learning Accuracy. In *AISTATS*, pages 113–120, 2010.
- [14] C. M. De Sa, C. Zhang, K. Olukotun, and C. Ré. Taming the wild: A unified analysis of hogwild-style algorithms. In *NIPS*, 2015.
- [15] J. Dean et al. Large scale distributed deep networks. In *NIPS*, 2012.
- [16] Y. Gong, L. Liu, M. Yang, and L. Bourdev. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*, 2014.
- [17] S. Gopi, P. Netrapalli, P. Jain, and A. V. Nori. One-Bit Compressed Sensing: Provable Support and Vector Recovery. In *ICML (3)*, pages 154–162, 2013.
- [18] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan. Deep Learning with Limited Numerical Precision. In *ICML*, pages 1737–1746, 2015.
- [19] S. Hadjis, C. Zhang, I. Mithiagkas, D. Iter, and C. Ré. Omnivore: An Optimizer for Multi-device Deep Learning on CPUs and GPUs. *ArXiv*, 2016.
- [20] S. Han et al. Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. *ArXiv*, 2015.
- [21] K. He et al. Deep Residual Learning for Image Recognition. *ArXiv*, 2015.
- [22] S. Hochreiter and J. Schmidhuber. *Neural Computation*, 1997.
- [23] I. Hubara et al. Quantized neural networks: Training neural networks with low precision weights and activations. *arXiv*, 2016.
- [24] F. N. Iandola et al. Firecaffe: near-linear acceleration of deep neural network training on compute clusters. In *CVPR*, 2016.
- [25] Y. Jia et al. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv*, 2014.
- [26] A. Krizhevsky et al. Learning multiple layers of features from tiny images, 2009.
- [27] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. *NIPS*, 2012.
- [28] B. Lesser et al. Effects of reduced precision on floating-point SVM classification accuracy. *Procedia Computer Science*, 4, 2011.
- [29] D. D. Lin, S. S. Talathi, and V. S. Annappureddy. Fixed point quantization of deep convolutional networks. *arXiv*, page, 2015.
- [30] Y. Low et al. Distributed GraphLab. *PVLDB*, 5(8):716–727, 2012.
- [31] X. Meng et al. MLib: machine learning in apache spark. *JMLR*, 2016.
- [32] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. University of Tennessee, 1994.
- [33] D. Miyashita, E. H. Lee, and B. Murmann. Convolutional neural networks using logarithmic data representation. *arXiv preprint arXiv:1603.01025*, 2016.
- [34] P. Moritz et al. SparkNet: Training Deep Networks in Spark. *ArXiv*, 2015.
- [35] A. Neelakantan et al. Adding gradient noise improves learning for very deep networks. *arXiv*, 2015.
- [36] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *ECCV*, 2016.
- [37] O. Russakovsky et al. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*, 115(3), 2015.
- [38] R. Schreier and G. C. Temes. *Understanding delta-sigma data converters*, volume 74. IEEE Press, Piscataway, NJ, 2005.
- [39] F. Seide et al. 1-bit stochastic gradient descent and application to data-parallel distributed training of speech dnns. In *Interspeech*, 2014.
- [40] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *ICLR*, 2015.
- [41] E. R. Sparks et al. KeystoneML: Optimizing Pipelines for Large-Scale Advanced Analytics. *ICDE*, 2017.
- [42] N. Strom. Scalable distributed DNN training using commodity GPU cloud computing. In *INTERSPEECH*, 2015.
- [43] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. Alemi. Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. *ArXiv*, 2016.
- [44] The Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *Arxiv*, 2016.
- [45] V. Vanhoucke et al. Improving the speed of neural networks on CPUs. In *Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop*, 2011.
- [46] W. Wang et al. Deep Learning At Scale and At Ease. *ArXiv*, 2016.
- [47] P. Xie et al. Lighter-communication distributed machine learning via sufficient factor broadcasting. In *UAI*, 2016.
- [48] M. Zaharia et al. Apache Spark. *CACM*, 2016.
- [49] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals. Understanding deep learning requires rethinking generalization. *ICLR*, 2017.
- [50] H. Zhang et al. ZipML: An End-to-end Bitwise Framework for Dense Generalized Linear Models. *ICML*, 2017.
- [51] S. Zhou et al. DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv*, 2016.

EasyCommit: A Non-blocking Two-phase Commit Protocol

Suyash Gupta, Mohammad Sadoghi
 Exploratory Systems Lab
 Department of Computer Science
 University of California, Davis

ABSTRACT

Large scale distributed databases are designed to support commercial and cloud based applications. The minimal expectation from such systems is that they ensure consistency and reliability in case of node failures. The distributed database guarantees reliability through the use of atomic commitment protocols. Atomic commitment protocols help in ensuring that either all the changes of a transaction are applied or none of them exist. To ensure efficient commitment process, the database community has mainly used the two-phase commit (2PC) protocol. However, the 2PC protocol is blocking under multiple failures. This necessitated the development of the non-blocking, three-phase commit (3PC) protocol. However, the database community is still reluctant to use the 3PC protocol, as it acts as a scalability bottleneck in the design of efficient transaction processing systems. In this work, we present Easy Commit which leverages the best of both the worlds (2PC and 3PC), that is, non-blocking (like 3PC) and requires two phases (like 2PC). Easy Commit achieves these goals by ensuring two key observations: (i) first transmit and then commit, and (ii) message redundancy. We present the design of the Easy Commit protocol and prove that it guarantees both safety and liveness. We also present a detailed evaluation of EC protocol, and show that it is nearly as efficient as the 2PC protocol.

1 INTRODUCTION

Large scale distributed databases have been designed and deployed for handling commercial and cloud-based applications [11, 14–18, 35, 37, 46–48, 56, 57]. The common denominator across all these databases is the use of transactions. A transaction is a sequence of operations that either reads or modifies the data. In case of geo-scale distributed applications, the transactions are expected to act on data stored in distributed machines spanning vast geographical locations. These geo-scale applications require the transactions to adhere to ACID [22] transactional semantics, and ensure that the database state remains consistent. The database is also expected to respect the atomicity boundaries that is either all the changes persist or none of the changes take place. In fact atomicity acts as a contract and establishes trust among multiple communicating parties. However, it is a common knowledge [39] that the distributed systems undergo node failures. Recent failures [19, 38, 55] have shown that the distributed systems are still miles away from achieving undeterred availability. In fact there is a constant struggle in the community to decide the appropriate level of database consistency and availability, necessary for achieving maximum system performance. The use of strong consistency semantics such as serializability [6] and linearizability [30] ensures system correctness. However, these properties have a causal effect on the underlying parameters such

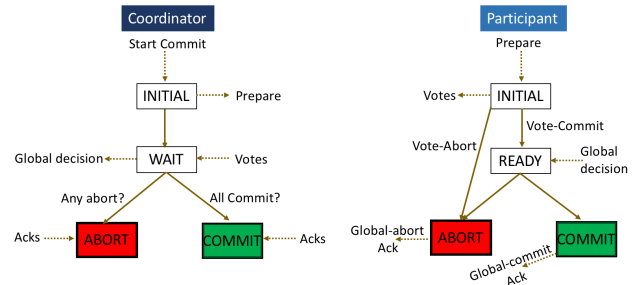


Figure 1: Two-Phase Commit Protocol

as latency and availability. Hence, a requirement for stronger consistency leads to a reduction in system availability.

There have been works that try to increase the database availability [2, 3]. But, more recently the distributed systems community has observed a shift in paradigm towards ensuring consistency. A large number of systems are moving towards providing strong consistency guarantees [15, 16, 18, 34, 41, 56]. Such a pragmatic shift has necessitated the use of agreement protocols such as Two-Phase Commit [21]. Commit protocols help in achieving the twin requirements of consistency and reliability in case of partitioned distributed databases. Prior research [9, 18, 42, 54, 56] has shown that data partitioning is an efficient approach to reduce contention and achieve high system throughput. However, a key point in hindsight is that the use of commit protocol should not be a cause for an increase in WAN communication latency in geo-scale distributed applications.

Transaction commit protocols help in reaching an agreement among the participating nodes when a transaction has to be committed or aborted. To initiate an agreement each participating node is asked to vote its decision on the operations on its transactional fragment. The participating nodes can decide to either commit or abort an ongoing transaction. In case of a node failure, the active participants take essential steps (run the termination protocol) to preserve database correctness.

One of the earliest and popular commitment protocol is the *two-phase commit* [21] (henceforth referred as 2PC) protocol. Figure 1 presents the state diagram [39, 52] representation of the 2PC protocol. This figure shows the set of possible states (and transitions) that a coordinating node¹ and the participating nodes follow, in response to a transaction commit request. We use solid lines to represent the state transitions and dotted lines to represent the inputs/outputs to the system. For instance, the coordinator starts the commit protocol on transaction completion, and requests all the participants to commence the same by transmitting *Prepare* messages. In case of multiple failures the two-phase commit protocol has been proved to be blocking [39, 51]. For example, if the coordinator and a participant fail, and if the remaining participants are in the READY state, then they cannot make progress (blocked!), as they are unaware about the state of the failed participant. This blocking characteristics

¹The coordinating node is the one which initiates the commit protocol, and in this work it is also the node which received the client request to execute the transaction.

of the 2PC protocol endangers database availability, and makes it unsuitable for use with the partitioned databases². The inherent shortcomings of the 2PC protocol led towards the design of resilient *three-phase commit* [50, 52] (henceforth referred as 3PC) protocol. The 3PC protocol introduces an additional PRE-COMMIT state between the READY and COMMIT states, which ensures that there is no direct transition between the non-committable and committable states. This simple modification makes the 3PC protocol non-blocking under node failures.

However, the 3PC protocol acts as the major performance suppressant in the design of efficient distributed databases. It can be easily observed that the addition of the PRE-COMMIT state leads to an extra phase of communication among the nodes. This violates the need of an efficient commit protocol for geo-scale systems. Hence, the design of a *hybrid* commit protocol, which leverages the best of both worlds (2PC and 3PC), is in order. We present the *Easy Commit* (a.k.a EC) protocol, which requires two phases of communication, and is non-blocking under node failures. We associate two key insights with the design of Easy Commit protocol that allow us to achieve the non-blocking characteristic in two phases. The first insight is to delay the commitment of updates to the database until the transmission of global decision to all the participating nodes, and the second insight is to induce message redundancy in the network. Easy Commit protocol introduces message redundancy by ensuring that each participating node forwards the global decision to all the other participants (including the coordinator). We now list down our contributions.

- We present the design of a new two-phase commit protocol and show it is non-blocking under node-failures.
- We also present an associated termination protocol, to be initiated by the active nodes, on failure of the coordinating node and/or participating nodes.
- We extend ExpoDB [45] framework to implement the EC protocol. Our implementation can be used seamlessly with various concurrency control algorithms by replacing 2PC protocol with EC protocol.
- We present a detailed evaluation of the EC protocol against the 2PC and 3PC protocol over two different OLTP benchmark suites: YCSB [10] and TPC-C [12], and scale the system upto 64 nodes, on the Microsoft Azure cloud.

The outline for rest of the paper is as follows: in Section 2, we motivate the need for EC protocol. In Section 3, we present design of the EC protocol. In Section 4, we present a discussion on assumptions associated with the design of commit protocols. In Section 5, we present the implementations of various commit protocols. In Section 6, we evaluate the performance of EC protocol against the 2PC and 3PC protocols. In Section 8, we present the related work, and conclude this work in Section 9.

2 MOTIVATION AND BACKGROUND

The state diagram representation for the two-phase commit protocol is presented in Figure 1. In 2PC protocol, the coordinator and participating nodes require at most two transitions to traverse from INITIAL state to the COMMIT or ABORT state. We use figure 3a to present the interaction between the coordinator and the participants, on a linear time scale. The 2PC commit protocol starts with the coordinator node transmitting a *Prepare* message

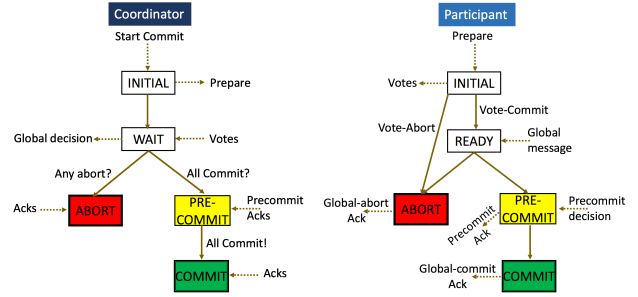


Figure 2: Three-Phase Commit Protocol

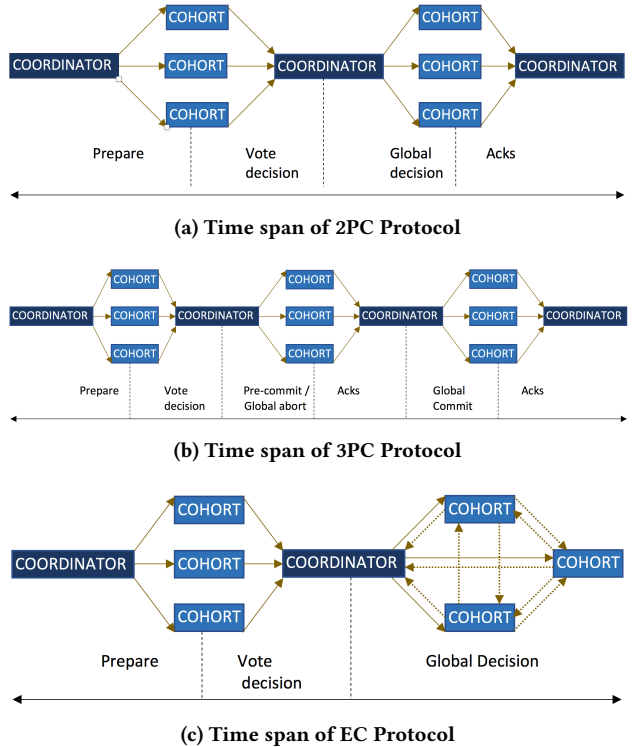


Figure 3: Commit Protocols Linearly Spanned

to each of the cohorts³ and adding a *begin_commit* entry in its log. When a cohort receives the *Prepare* message, it adds a *ready* entry in its log, sends its decision (*Vote-commit* or *Vote-abort*) to the coordinator. If a cohort decides to abort the transaction then it independently moves to the ABORT state, else it transitions to the READY state. The coordinator waits for the decision from all the cohorts. On receiving all the responses, it analyzes all the votes. If there is a *Vote-abort* decision, then the coordinator adds an *abort* entry in the log, transmits the *Global-Abort* message to all the cohorts and moves to the ABORT state. If all the votes are to commit, then the coordinator transmits the *Global-Commit* message to all the cohorts, and moves to COMMIT state, after adding a *commit* entry to log. The cohorts on receiving the coordinator decision move to the ABORT or COMMIT state, and add the *abort* or *commit* entry to the log, respectively. Finally, the cohorts acknowledge the global decision, which allows the coordinator to mark the completion of commit protocol.

²Partitioned database is the terminology used by the database community to refer to the shared-nothing distributed databases, and should not be intermixed with the term network partitioning.

³The term cohort refers to a participating node in the transaction commit process. We use these terms interchangeably.

The 2PC protocol has been proved to be blocking [39, 51] under multiple node failures. To illustrate this behavior let us consider a simple distributed database system with a coordinator C and three participants X , Y and Z . Now assume a snapshot of the system when C received *Vote-commit* from all the participants, and hence, it decides to send *Global-Commit* message to all the participants. However, say C fails after transmitting *Global-Commit* message to X , but before sending messages to Y and Z . The participant X on receiving the *Global-Commit* message, commits the transaction. Now, assume X fails after committing the transaction. On the other hand, nodes Y and Z would *timeout* due to no response from the coordinator, and would be blocked indefinitely, as they require node X to reach an agreement. They cannot make progress, as neither they have knowledge of the global decision nor they know the state of node X before failure. This situation can be prevented with the help of the three-phase commit protocol [50, 52].

Figure 2 presents the state transition diagram for the coordinator and cohort executing the three-phase commit protocol, while figure 3b expands the 3PC protocol on the linear time scale. In the first phase, the coordinator and the cohorts, perform the same set of actions as in the 2PC protocol. Once the coordinator checks all the votes, it decides whether to abort or commit the transaction. If the decision is to abort, the remaining set of actions performed by the coordinator (and the cohorts) are similar to the 2PC protocol. However, if the coordinator decides to commit the transaction, then it first transmits a *Prepare-to-Commit* message, and adds a *pre-commit* entry to the log. The cohorts on receiving the *Prepare-to-Commit* message, move to the PRE-COMMIT state, add a corresponding *pre-commit* entry to the log, and acknowledge the message reception to the coordinator. The coordinator then sends a *Global-Commit* message to all the cohorts, and the remaining set of actions are similar to the 2PC protocol.

The key difference between the 2PC and 3PC protocol is the PRE-COMMIT state, which makes the latter non-blocking. The design of 3PC protocol is based on the Skeen’s [50] design of a non-blocking commit. In his work Skeen laid down two fundamental properties for the design of a non-blocking commit protocol: (i) no state should be adjacent to both the ABORT and COMMIT states, and (ii) no non-committable⁴ state should be adjacent to the COMMIT state. These requirements motivated Skeen to introduce the notion of a new committable state (PRE-COMMIT) to the 2PC state transition diagram.

The existence of PRE-COMMIT state makes the 3PC protocol non-blocking. The aforementioned multi-node failure case does not indefinitely block the nodes Y and Z which are waiting in the READY state. The nodes Y and Z can make safe progress (by aborting the transaction) as they are assured that the node X could not have committed the transaction. Such a behavior is implied by the principle that no two nodes could be more than one state transition apart. The node X is guaranteed to be in one of the following states: INITIAL, READY, PRE-COMMIT and ABORT, at the time of failure. This indicates that node X could not have committed the transaction, as nodes Y and Z are still in the READY state (It is important note that in the 3PC protocol the coordinator sends the *Global-Commit* message after it transmits the *Prepare-to-commit* message to all the nodes.). Interestingly, if either of nodes Y or Z are in the PRE-COMMIT state then they can actually commit the transaction. However, it can be easily observed that the non-blocking characteristic of the 3PC protocol comes at an additional cost, an extra round of handshaking.

⁴INITIAL, READY and WAIT states are considered as non-committable states.

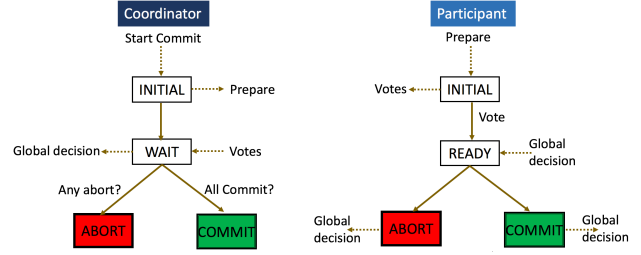


Figure 4: Easy Commit Protocol

3 EASY COMMIT

We now present the *Easy Commit (EC)* protocol. EC is a two-phase protocol, but unlike 2PC it exhibits non-blocking behavior. The EC protocol achieves these goals through two key insights: (i) first transmit and then commit, and (ii) message redundancy. In the second phase, Easy Commit ensures that each participating node forwards the coordinating node’s decision to all the other participants. To ensure non-blocking behavior, EC protocol also requires each node (coordinator and participants) to transmit the global decision to all the other nodes, before they commit. Hence, the commit step subsumes message transmission to all the nodes.

3.1 Commitment Protocol

We present the EC protocol state transition diagram, and the coordinator and participant algorithms in Figure 4 and Figure 5, respectively. The EC protocol is initiated by the coordinator node. It sends the *Prepare* message to each of the cohorts and moves to the READY state. When a cohort receives the *Prepare* message, it sends its decision to the coordinator, and moves to the READY state. On receiving the responses from each of the cohorts, the coordinator first transmits the global decision to all the participants, and then commits (or aborts) the transaction. Each of the cohorts, on receiving a response from the coordinator, first forward the global decision to all the participants (and the coordinator), and then commit (or abort) the transaction locally.

We introduce multiple entries to the log to facilitate recovery during node failures. Note: the EC protocol allows the coordinator to commit as soon as it has communicated the global decision to all the other nodes. This implies that the coordinator need not wait for the acknowledgments. When a node timeouts, while waiting for a message, it executes the *termination protocol*. Some of the noteworthy observations are:

- I. A participant node cannot make a direct transition from the INITIAL state to the ABORT state.
- II. The cohorts, irrespective of the global decision, always forward it to every participant.
- III. The cohorts need not wait for message from the coordinator, if they receive global decision from other participants.
- IV. There exists some hidden states (a.k.a TRANSMIT-A and TRANSMIT-C), only after which a node aborts or commits the transaction (cf. discussed in Section 3.2).

In Figure 3c, we also present the linear time scale model for the Easy Commit protocol. Here, in the second phase, we use solid lines to represent the global decision from the coordinator to the cohorts, and the dotted lines to represent message forwarding.

```

Send Prepare to all participants;
Add begin_commit to log;
Wait for (Vote-commit or Vote-abort) from all participants;
if timeout then
  Run Termination Protocol;
end if
if All messages are Vote-commit then
  Add global-commit-decision-reached in log;
  Send Global-commit to all participants;
  Commit the transaction;
  Add transaction-commit to log;
else
  Add global-abort-decision-reached in log;
  Send Global-abort to all participants;
  Abort the transaction;
  Add transaction-abort to log;
end if

```

(a) Coordinator's algorithm

```

Wait for Prepare from the coordinator;
if timeout then
  Run Termination Protocol;
end if
Send decision (Vote-commit or Vote-abort) to coordinator;
Add ready to log;
Wait for message from coordinator;
if timeout then
  Run Termination Protocol;
end if
if Coordinator decision is Global-commit then
  Add global-commit-received in log;
  Forward Global-commit to all nodes;
  Commit the transaction;
  Add transaction-commit to log;
else
  Add global-abort-received in log;
  Forward Global-abort to all nodes;
  Abort the transaction;
  Add transaction-abort to log;
end if

```

(b) Participant's algorithm

Figure 5: Easy Commit Algorithm.

3.2 Termination Protocol

We now consider the correctness of the EC algorithm under *node-failures*. We want to ensure that the EC protocol exhibits both liveness and safety properties. A commit protocol is said to be *safe* if there isn't any instant during the execution of the system under consideration when two or more nodes are in conflicting states (that is one node is in COMMIT state while other is in ABORT). A protocol is said to respect *liveness* if its execution causes none of the nodes to block.

During the execution of a commit protocol, each node waits for a message for a specific amount of time before it *times out*. When a node times out then it concludes loss of communication with the sender node, which in our case corresponds to failure of the sender node. A node is assumed to be blocked if it is unable to make progress on timeout. In case of such node failures, the active nodes execute the termination protocol to ensure system

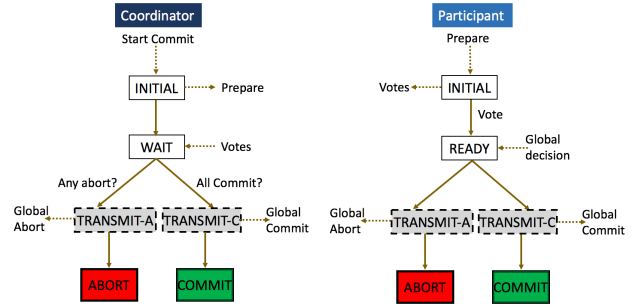


Figure 6: Logical expansion of Easy Commit Protocol.

makes progress. We illustrate the termination protocol by stating the actions taken by the coordinator and the participating nodes on timeout. The coordinator can timeout only in the WAIT state, while the cohorts can timeout in INITIAL and READY states.

- A. **Coordinator Timeout in WAIT state** – If the coordinator times out in this state, then it implies that the coordinator didn't receive the vote from one of the cohorts. Hence, the coordinator first adds a log entry for the *global-abort-decision-reached*, then transmits the *Global-abort* message to all the active participants, and finally aborts the transaction.
- B. **Cohort Timeout in INITIAL State** – If the cohort times out in this state, then it implies that it didn't receive the *Prepare* message from the coordinator. Hence, this cohort communicates with other active cohorts to reach a common decision.
- C. **Cohort Timeout in READY State** – If the cohort times out in this state, it implies that it didn't receive a *Global-Commit* (or *Global-Abort*) message from any node. Hence, it would consult the active participants to reach a decision common to all the participants.

Leader Election: In last two cases we force the cohorts to perform transactional commit or abort based on an agreement. This agreement requires selection of a new leader (or coordinator). The target of this leader is to ensure that all the active participants, follow the same decision that is commit (or abort) the transaction. The selected leader can be in the INITIAL or the WAIT state. It consults all the nodes if any of them has received a copy of the global decision. If none of the nodes know the global decision, then the leader first adds a log entry for the *global-abort-decision-reached*, then transmits the *Global-abort* message to all the active participants, and finally aborts the transaction.

To prove correctness of EC protocol, Figure 6 expands the state transition diagram. We introduce two intermediate hidden states (a.k.a TRANSMIT-A and TRANSMIT-C). All the nodes are oblivious to these states, and the purpose of these states is to ensure message redundancy in the network. As a consequence, we can categorize the states of the EC protocol under five heads:

- UNDECIDED – The state before reception of global decision (that is INITIAL, READY and WAIT states).
- TRANSMIT-A – The state on receiving the global abort.
- TRANSMIT-C – The state on receiving the global commit.
- ABORT – The state after transmitting *Global-Abort*.
- COMMIT – The state after transmitting *Global-Commit*.

Figure 7 illustrate whether two states can co-exist (Y) or they conflict (N). We derive this table on the basis of our observations: I - IV and cases A - C. We now have sufficient tools to prove the liveness and safety property of EC protocol.

	UNDECIDED	T-A	T-C	ABORT	COMMIT
UNDECIDED	Y	Y	Y	N	N
T-A	Y	Y	N	Y	N
T-C	Y	N	Y	N	Y
ABORT	N	Y	N	Y	N
COMMIT	N	N	Y	N	Y

Figure 7: Coexistent states in EC protocol (T-A refers to TRANSMIT-A and T-C refers to TRANSMIT-C).

THEOREM 3.1. *Easy Commit protocol is safe, that is in the presence of only node failures, for a specific transaction, two nodes cannot be in both Aborted and Committed states, at any instant.*

PROOF. Let us assume the case that two nodes p and q are in the conflicting states (say p voted to abort the transaction and q voted to commit). This would imply that one of them received *Global-Commit* message while the other received *Global-Abort*. From (II) and (III) we can deduce that p and q should transmit the global decision to each other, but as they are in different states, it implies a contradiction. Also, from (I) we have the guarantee that p could not have directly transitioned to the ABORT state. This implies p and q would have received message from some other node. But, then they should have received the same global decision.

Hence, we assume that either of the nodes p or q moved to a conflicting state and then failed. But, this violates property (IV) which states that a node needs to transmit its decision to all the other nodes before it can commit or abort the transaction. Also, once either of p or q fails, the rest of the system follows termination protocol (cases (A) to (C)), and reaches a safe state. It is important to see that the termination protocol is re-entrant. \square

THEOREM 3.2. *Easy Commit protocol is live that is in the presence of only node failures, it does not block.*

PROOF. The proof for this theorem is a corollary of Theorem 3.1. The termination protocol cases (A) to (C) provide the guarantee that the nodes do not block and can make progress, in case of a node failure. \square

3.3 Comparison with 2PC Protocol

We now draw out comparisons between the the 2PC and EC protocols. Although, EC protocol is non-blocking, but still 2PC protocol has a lower message complexity. EC protocol’s message complexity is $O(n^2)$, while the message complexity for 2PC $O(n)$.

To illustrate the non-blocking property of EC protocol, we now tackle the motivational example of multiple failures. For the sake of completeness we restate the example here. Let us assume a distributed system with coordinator C and participants X , Y and Z . We also assume that C decides to transmit *Global-Commit* message to all the nodes, and fails just after transmitting message to the participant X . Say, the node X also fails after receiving the message from C . Thus, nodes Y and Z neither received messages from C nor from node X . In this setting, the nodes Y and Z would eventually timeout, and run the termination protocol. From case (C) of termination protocol, it is evident that the nodes Y and Z would select a new leader among themselves, and would safely transit to the ABORT state.

3.4 Comparison with 3PC protocol

Although, EC protocol looks similar to 3PC protocol, but it is a stricter and an efficient variant to 3PC protocol. It introduces

the notion of a set of intermediate hidden states: TRANSMIT-A and TRANSMIT-C, which can be superimposed on the ABORT and COMMIT states, respectively. Also, in the EC protocol, the nodes do not expect any acknowledgements. So unlike the 3PC protocol, there are no inputs to the TRANSMIT-A, TRANSMIT-C, ABORT and COMMIT states. However, EC protocol has a higher message complexity than 3PC, which has a message complexity of $O(n)$.

4 DISCUSSION

Until now, all our discussion assumed existence of only node failures. In Section 3 we prove that EC protocol is non-blocking under node failures. We now discuss the behavior of the 2PC, 3PC and EC protocols under communication failures that is message delay and message loss. Later in this section we also study the degree to which these protocols support independent recovery.

4.1 Message Delay and Loss

We now analyze the characteristics of 2PC, 3PC and EC protocol, under unexpected delays in message transmission. Message delays represent an unprecedented lag in the communication network. The presence of message delays can cause a node to timeout and act as if a node failure has occurred. This node may receive a message pertaining transaction commitment or abort, after the decision has been made. It is interesting to note that 2PC and 3PC protocols are not safe under message delays [7, 39]. Prior works [26, 39] have shown that it is impossible to design a non-blocking commitment protocol for unbounded asynchronous networks with even a single failure.

We illustrate the nature of 3PC protocol under message delay, as it is trivial to show that 2PC protocol is unsafe under message delays. The 3PC protocol state diagram does not provide any intuition about the transitions that two nodes should perform when both of them are active but unable to communicate. In fact, partial communication or unprecedented delay in communication can easily hamper the database consistency.

Let us consider a simple configuration with a coordinator C and the participants X , Y and Z . Now assume that C receives *Vote-commit* message from all the cohorts, and, hence it decides to send the *Prepare-to-Commit* message to all the cohorts. However, it is possible that the system starts facing unanticipated delays on all the communication links with C at one end. We can also assume that the paths to node X are also facing severe delays. In such a situation, the coordinator would proceed to *globally commit* the transaction (as it has moved to the PRE-COMMIT state), while the nodes X , Y and Z would abort the transaction (as from their perspective the system has undergone multiple failures). This implies that the 3PC termination protocol is not sound under message delays, and similarly we can show that EC protocol is unsafe under message delays.

This situation can aggravate if the network undergoes message loss. Interestingly, message loss has been deemed to be true representation of the network partitioning [39]. Hence, no commit protocol is safe (or non-blocking) under message loss [7]. If the system is suffering from message loss then the participating nodes (and coordinator) would *timeout*, and would run the associated terminating protocol that could make nodes transit to conflicting states. Thus, we also conclude that 2PC, 3PC and EC protocols are unsafe under message loss.

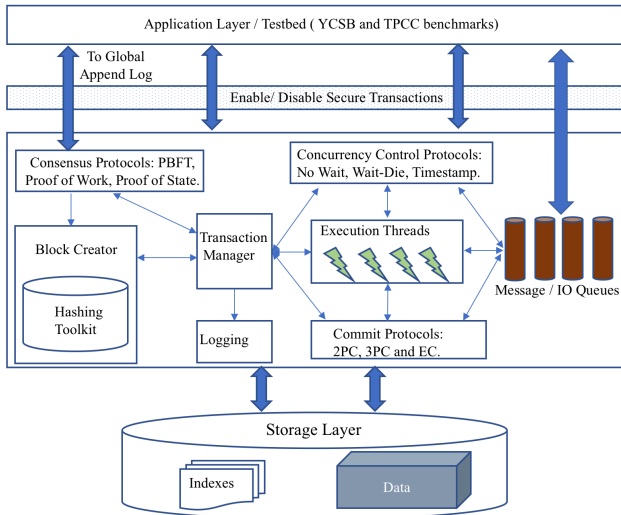


Figure 8: ExpoDB Framework - executed at each server process, hosted on a cloud node. Each server process receives a set of messages (from clients and other servers), and uses multiple threads to interact with various distributed database components.

4.2 Independent Recovery

Independent recovery is one of the desired properties from the nodes in a distributed system. An independent recovery protocol lays down a set of rules that help a failed node to terminate (commit or abort) the transaction which was executing at the time of its failure, without any help from other active participants. It is interesting to note that 2PC and 3PC protocols only support partial independent recovery [7, 39].

It is easy to present a case where the 3PC protocol lacks independent recovery. Consider a cohort in the READY state that votes to commit the transaction, and fails. On recovery this node needs to consult with the other nodes about the fate of the last transaction. This node cannot independently commit (or abort) the transaction, as it does not know the global decision, which could have been either commit or abort.

EC protocol supports independent recovery in following cases:

- (i) If a cohort fails before transmitting its vote, then on recovery it can simply abort the transaction.
- (ii) If the coordinator fails before transmitting the global decision then it aborts the transaction on recovery.
- (iii) If either coordinator or participant fail after transmitting the global decision and writing the log, then on recovery they can use this entry to reach the consistent state.

5 EASY COMMIT IMPLEMENTATION

We now present a discussion on our implementation of the Easy Commit (EC) protocol. We have implemented EC protocol in the ExpoDB platform [45]. ExpoDB is an in-memory, distributed transactional platform that incorporates and extends the Deneva [28] testbed. ExpoDB also offers secure transactional capability, and presents a flexible framework to study distributed ledger—blockchain [8, 43].

5.1 Architectural Overview

ExpoDB includes a lightweight layer for testing distributed protocols and design strategy. Figure 8 presents the block diagram

representation of the ExpoDB framework. It supports a client-server architecture, where each client or server process is hosted on one of the cloud nodes. To maintain inherent characteristics of a distributed system, we opt for a shared nothing architecture. Each partition is mapped to one server node.

A transaction is expressed as a stored procedure that contains both program logic and database queries, which read or modify the records. The clients and server processes communicate with each other using TCP/IP sockets. In practice, the client and server processes are hosted on different cloud nodes, and we maintain an equal number of client and server cloud instances.

Each client creates one or more transactions, and sends the transaction execution request to the server process. The server process in turn executes the transaction by accessing the local data and runs the transaction until further execution requires access to remote data. The server process then communicates with other server processes that have access to remote data (remote partitions), to continue the execution. Once, these processes return the result, the server process continues execution till completion. Next, it takes a decision to commit or abort the transaction (that is, executes the associated *commit protocol*).

In case a transaction has to be aborted then the coordinating server sends messages to the remote servers to rollback the changes. Such a transaction is resumed after an exponential back-off time. On successful completion of a transaction, the coordinating server process sends an acknowledgment to the client process, and performs necessary garbage collection.

5.2 Design of 2PC and 3PC

2PC: The 2PC protocol starts after the completion of the transaction execution. The read-only transactions and single partition transactions do not make use of the commit protocol. Hence, the commit protocol comes into play when the transaction is multi-partition and performs updates to the data-storage. The coordinating server sends a *Prepare* message to all the participating servers, and waits for their response. The participating servers respond with the *Vote-commit* message⁵. On receiving the *Vote-commit* message the coordinating server starts the final phase, and transmits the *Global-Commit* message to all the participants. Each participant on receiving the *Global-Commit* message commits the transaction, releases the local transactional resources, and responds with an acknowledgment for the coordinator. The coordinator waits on a counter for response from each participant and then commits the transaction, sends a response to the client node, and releases the associated transactional data-structures.

3PC: To gauge the performance of the EC protocol, we also implemented the three-phase commit protocol. The 3PC protocol implementation is a straightforward extension to the 2PC protocol. We add an extra PRE-COMMIT phase before the final phase. On receiving, all the *Vote-commit* messages, the coordinator sends the *Precommit* message to each participant. The participating nodes acknowledge the reception of the *Precommit* message from the coordinator. The coordinating server on receiving these acknowledgments, starts the finish phase.

5.3 Easy Commit Design

We now explain the design of Easy Commit protocol in the ExpoDB framework. The first phase (that is the INITIAL phase) is same for both the 2PC and the EC protocol. In the EC protocol,

⁵Without node failures, any transaction that reaches the prepare phase is assumed to successfully commit.

once the coordinator receives the *Vote-commit* message from all the nodes, it first sends the *Global-commit* message to each of the participating processes, and then commits the transaction. Next it, responds to the client with the transaction completion notification. When the participating nodes receive the *Global-Commit* message from the coordinator, they forward the *Global-Commit* message to all the other nodes (including the coordinator), and then commit the transaction.

Although, in the EC protocol the coordinator has a faster response rate to the client, but its throughput takes a slight dip due to additional, implementation enforced wait. It can be noted that we have not performed any cleanup tasks (such as releasing the transactional resources) yet. The cleanup of the transactional resources is performed once it is ensured that neither of those resources would be ever used, nor any messages associated with the transaction would be further received. Hence, we have to force all the nodes (both the coordinator and the participants) to poll the message queue, and wait till they have received the messages from each other node. Once all the messages are received, each node performs the cleanup.

To implement EC protocol we had to extend the message being transmitted with a new field which identifies all the participants of the transaction. This array contains the Id for each participant, and is updated by the coordinator (as only the coordinator has information about all the partitions) and transmitted as part of the *Global-Commit* message.

6 EVALUATION

In this section, we present a comprehensive evaluation of our novel Easy Commit protocol against 2PC and 3PC. As discussed in Section 5, we use the ExpoDB framework for implementing the EC protocol. For our experimentation, we adopt the evaluation scheme of Harding et al. [28].

To evaluate various commit protocols, we deploy the ExpoDB framework on the Microsoft Azure cloud. For running the client and server processes, we use upto 64 Standard_D8S_V3 instances, deployed in the US East region. Each Standard_D8S_V3 instance consists of 8 virtual CPU cores and 32GB of memory. For our experiments, we ensure a one-to-one mapping between the server (or client) process and the hosting Standard_D8S_V3 instance. On each server process, we allowed creation of 4 worker threads, each of which were attached to a dedicated core, and 8 I/O threads. At each server node, a load of 10000 open client connections is applied. For each experiment, we first initiated a warmup phase for 60 seconds, followed by 60 seconds of execution. The measured throughput does not include the transactions completed during warmup phase. If a transaction gets aborted then it is restarted again, only after a fixed time. To attenuate the noise in our readings, we average our results over three runs.

To evaluate the commit protocols, we use the NO_WAIT concurrency control algorithm. We use the NO_WAIT algorithm as: (i) it is the simplest algorithm, amongst all the concurrency control algorithms present in the ExpoDB framework, and (ii) has been proved to achieve high system throughput. It has to be noted that the use of underlying concurrency control algorithm is orthogonal to our approach. We present the design of a new commit protocol, and hence other concurrency control algorithms (except Calvin) available in the ExpoDB framework, can also employ EC protocol during the commit phase. We present a discussion on the different concurrency control algorithms, later in this section.

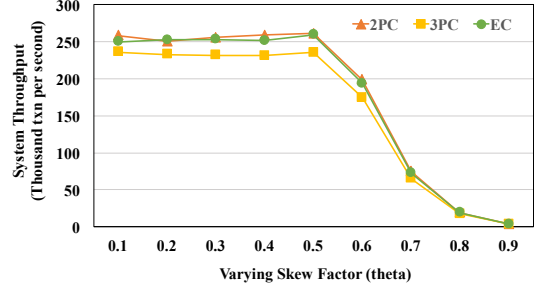


Figure 9: System throughput (transactions per second) on varying the skew factor (theta) for the 2PC, 3PC and EC protocols. These experiments run the YCSB benchmark. Number of server nodes are set to 16 and partitions per transaction are set to 2.

In NO_WAIT protocol, a transaction requesting access to a locked record is aborted. On aborting the transaction, all the locks held with this transaction are released, which allows other transactions waiting on these locks to progress. NO_WAIT algorithm prevents deadlock by aborting transactions in case of conflicts, and hence, has high abort rate. The simple design of NO_WAIT algorithm, and its ability to achieve high system throughput [28] motivated us to use it for concurrency control.

6.1 Benchmark Workloads

We test our experiments on two different benchmark suites: YCSB [10] and TPC-C [12]. We use YCSB benchmark to evaluate EC protocol on characteristics interesting to the OLTP database designers (Section 6.2 to Section 6.5), and use TPC-C to gauge the performance of EC protocol from the perspective of a real world application (Section 6.6 and Section 6.7).

YCSB – The Yahoo! Cloud Serving Benchmark consists of 11 columns (including a primary key) and 100B random characters. In our experiments we used a YCSB table of size 16 million records per partition. Hence, the size of our database was 16 GB per node. For all our experiments we ensured that each YCSB transaction accessed 10 records (we mention changes to this scheme explicitly). Each access to YCSB data followed the Zipfian distribution. Zipfian distribution tunes the access to hot records through the *skew factor* (theta). When theta is set to 0.1, the resulting distribution is uniform, while the theta value 0.9 corresponds to extremely skewed distribution. In our evaluation using YCSB data, we only executed multi-partition transactions, as single partition transactions do not require use of commit algorithms.

TPC-C – The TPC-C benchmark helps to evaluate system performance by modeling an application for warehouse order processing. It consists of a read-only, item table that is replicated at each server node while rest of the tables are partitioned using the warehouse ID. ExpoDB supports *Payment* and *NewOrder* transactions, which constitute 88% of the workload. Each transaction of *Payment* type accesses at most 2 partitions. These transaction first update the payment amounts for the local warehouse and district, and then update the customer data. The probability that a customer belongs to a remote warehouse is 0.15. In case of transactions of type *NewOrder*, first the transaction reads the local warehouse and district records and then modifies the district record. Next, it modifies item entries in the stock table. Only, 10% *NewOrder* transactions are multi-partition, as only 1% of the updates require remote access.

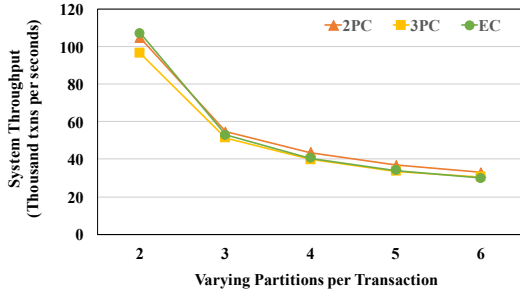


Figure 10: System throughput (transactions per second) on varying the number of partitions per transactions for the commit protocols. These experiments use YCSB benchmark. The number of server nodes are set to 16 and theta is set to 0.6.

6.2 Varying Skew factor (Theta)

We evaluate the system throughput by tuning the skew factor (theta), available in YCSB benchmarks, from 0.1 to 0.9. Figure 9 presents the statistics when the number of partitions per transaction are set to 2. In this experiment, we use 16 server nodes to analyze the effects induced by the three commit protocols.

A key takeaway from this plot is that, for $\theta \leq 0.7$ the system throughputs for EC and 2PC protocols are better than the system throughput for the 3PC protocol. On increasing the theta further the transactional access becomes highly skewed. This results in an increased contention between the transactions as they try to access (read or write) the same record. Hence, there is a significant reduction in the system throughput across various commit protocols. Thus, it can be observed that the magnitude of difference in the system throughputs for 2PC, 3PC and EC protocol is relatively insignificant. It is important to note that on highly skewed data, the gains due to the choice of underlying commit protocols are overshadowed by other system overheads (such as cleanup, transaction management and so on).

In the YCSB benchmark, for $\theta \leq 0.5$ the data access is uniform across the nodes, which implies that the client transactions access data on various partitions – low contention. Hence, each server node achieves nearly the same throughput. It can be observed that for all the three commit protocols the throughput is nearly constant (not same). We attribute the delta difference in the throughputs of the EC and 2PC protocols to the system induced overheads, network communication latency, and resource contention between the threads (for access to cpu and cache).

6.3 Varying Partitions per Transaction

We now measure the system throughput achieved by the three commit protocols on varying the number of partitions per transactions from 2 to 6. Figure 10 presents the throughput achieved on the YCSB benchmark, when theta is fixed to 0.6, and number of server nodes are set to 16. The number of operations accessed by each transaction are set to 16, and the transaction read-write ratio is maintained at 1 : 1.

It can be observed that on increasing the number of partitions per transaction there is a dip in the system throughput, across all of the commit protocols. On moving from 2 to 4 partitions there is an approximate decrease of 55%, while the reduction in system performance is around 25% from 4 partitions to 6 partitions, for the three commit protocols. As the number of partitions per transaction increase, the number of messages being exchanged in each round increases linearly for 2PC and 3PC, and quadratically for EC. Also, an increase in partitions imply the transactional

resources are held longer across multiple sites, which leads to throughput degradation for all the protocols. Note: in practice, the number of partitions per transaction are not more than four [12].

6.4 Varying Server Nodes

We study the effect of varying the number of server nodes (from 2 nodes to 32 nodes) on the system throughput and latency, for the 2PC, 3PC and EC protocols. In Figure 11 we set the number of partitions per transaction to 2 and plot graphs for the low contention ($\theta = 0.1$), medium contention ($\theta = 0.6$) and high contention ($\theta = 0.7$). In these experiments, we increase size of YCSB table in accordance to the increase in number of server nodes.

In Figure 11, we use the plots on the left to study the system throughput on varying the number of server nodes. It can be observed that as the contention (or skew factor) increases the system throughput decreases, and such a reduction is sharply evident on moving from $\theta = 0.6$ to $\theta = 0.7$. Another interesting observation is that the system throughput attained by the EC protocol is significantly greater than the throughput attained under 3PC protocol. The gains in system throughput are due to reduction of an extra phase which compensates for the extra messages communicated during the EC protocol.

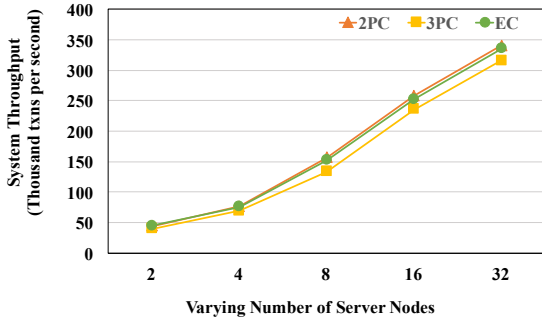
In comparison to the 2PC protocol the system throughput under EC protocol is marginally lower at low contention and medium contention, and relatively same at high contention. These gains are the result of zero acknowledgment messages required by the coordinating node, in the commit phase, which helps EC protocol perform nearly as efficient as the 2PC protocol. This helps us to conclude that a database system using EC is as scalable as its counterpart employing 2PC.

6.4.1 Latency. In Figure 11, we use the plots on the right, to shows the 99 percentile system latency when one of the three commit protocols are employed by the system. We again vary the number of server nodes from 2 to 32. The 99 percentile latency is measured from the first commit to the final commit of a transaction. On increasing the number of server nodes there is a steep increase in latency for each commit protocol. The high latency values for 3PC protocol can be easily cited to the extra phase of communication.

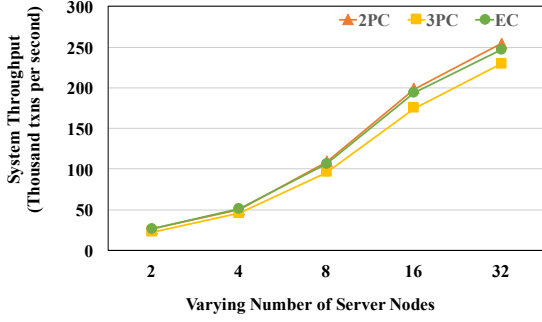
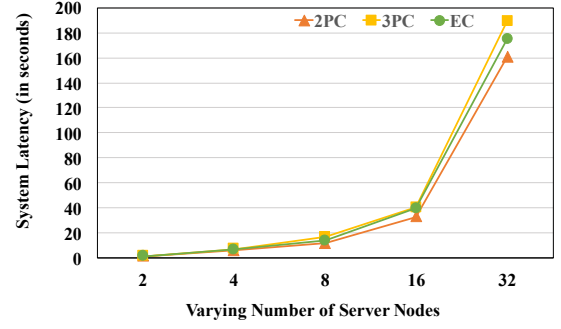
6.4.2 Proportion of time consumed by various components: Figure 12 presents the time spent on various components of the distributed database system. We show the time distribution for the different degree of contention (θ). We categorize these measures under seven different heads.

Useful Work is the time spent by worker threads doing computation for read and write operations. **Txn Manager** is the time spent in maintaining transaction associated resources. **Index** is the time spent in transaction indexing. **Abort** is the time spent in cleaning up aborted transactions. **Idle** is the time worker thread spends when not performing any task. **Commit** is the time spent in executing the commit protocol. **Overhead** represents the time to fetch transaction table, transaction cleanup and releasing transaction table.

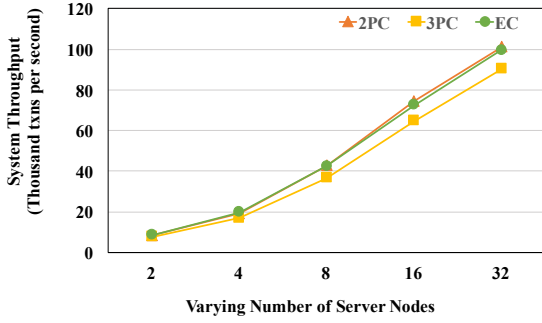
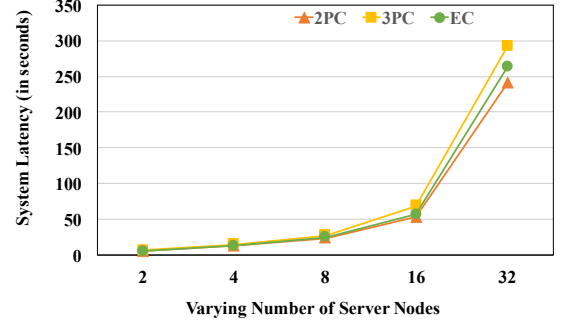
The key intuition from these plots is that as the contention (theta) increases there is an increase in time spent in abort. At low contention as most of the transactions are read-only, so the time spent in commit phase is least, and as contention increase, commit phase plays an important role in achieving high throughput from databases. Also, it can be observed at medium and high contention, worker threads executing 3PC protocol



(a) Low contention – (theta = 0.1).



(b) Medium contention – (theta = 0.6).



(c) High contention – (theta = 0.7).

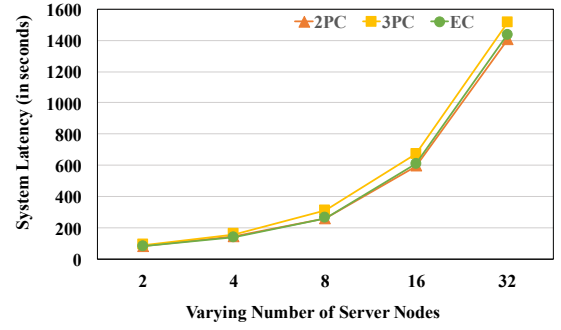


Figure 11: System Throughput (transactions per second) and System Latency (in seconds), on varying the number of server nodes for the 2PC, 3PC and EC protocols. The measured latency is the 99-percentile latency, that is, latency from the first start to final commit of a transaction. For these experiments we use the YCSB benchmarks and set the number of partitions per transaction to 2.

are idle for the maximum time and perform the least amount of useful work, which indicates a decrease in system throughput under 3PC protocol due to an extra phase of communication.

6.5 Varying Transaction Write Percentage

We now vary the transactional write percentage, and draw out comparisons between the system throughput achieved by the Ex-poDB when employing one of the three commit protocols. These experiments are based on YCSB benchmark, and vary the percentage of write operations accessed by each transaction from 10 to 90. We set the skew factor to 0.6, number of server nodes to 16 and partitions per transaction to 2.

It can be seen that when only 10% of the operations are write then all the protocols achieve nearly the same system throughput. This is because most of the requests sent by the client consists of read-only transactions, and under read only transactions, the commit protocols are not executed. However, as the write percentage increases the gap between the system throughput achieved by 3PC protocol and the other two commit protocols increases. This indicates that 3PC protocol performs poorly when the underlying application consists of write intensive transactions.

In comparison to the 2PC protocol, EC protocol undergoes marginal reduction in throughput. As the number of write operations increase, the number of transactions undergoing the commit protocol also increase. We have already seen that under EC protocol (i) the amount of message communication is higher than the 2PC protocol, and (ii) each node needs to wait for additional *wait-time* before releasing the transactional resources. Some of these held resources include locks on data items, and it is easy to surmise that under EC protocol locks are held longer than the 2PC protocol. The increase in duration of locks being held also leads to an increased abort rate, which is another important factor for reduced system throughput.

6.6 Scalability of TPC-C Benchmarks

We now gauge the performance of the EC protocol with respect to a real-world application, that is using TPC-C benchmark. Figure 14 presents the characteristics of the 2PC, 3PC and EC protocols, under TPC-C benchmark, on varying the number of server nodes. It has to be noted that a major chunk of TPC-C transactions are single-partition, while most of the multi-partition transactions access only two partitions. Our evaluation scheme

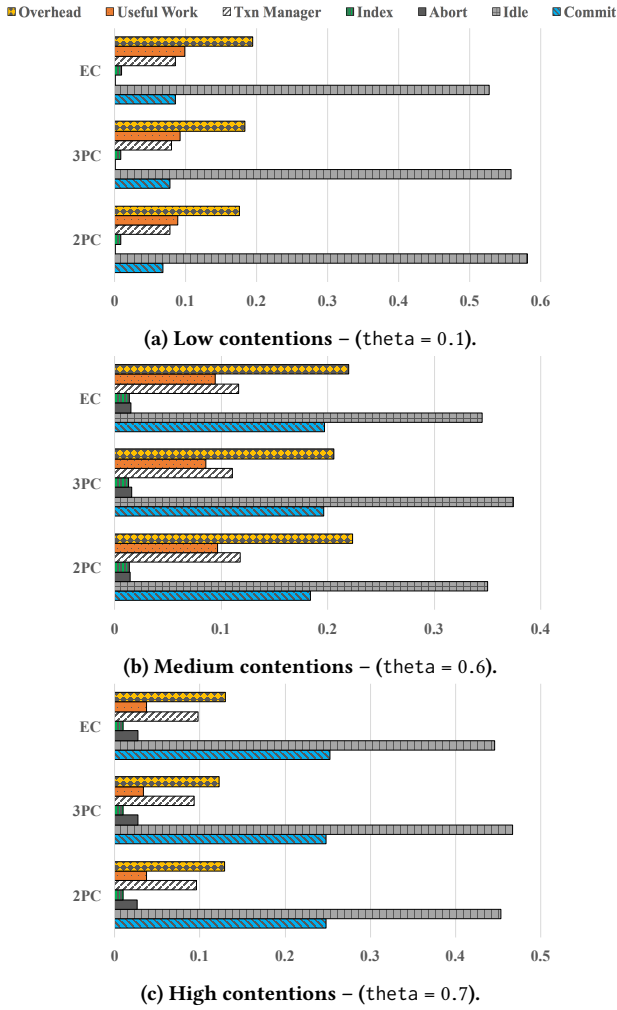


Figure 12: Percentage of time spent by various database components, on executing the YCSB benchmark. We set the number of server nodes to 16 and partitions per transaction to 2.

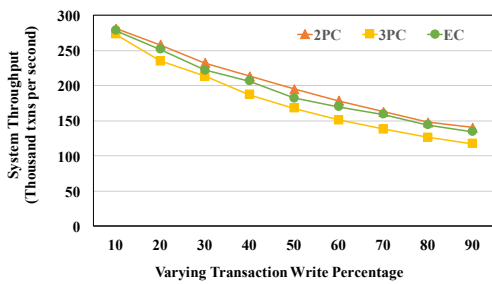
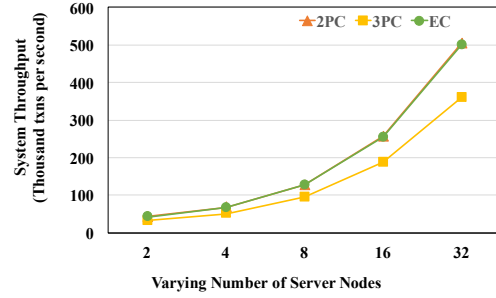


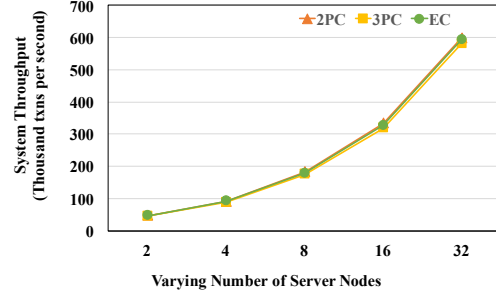
Figure 13: System throughput (transactions per second) on varying the transaction write percentage for the 2PC, 3PC and EC protocols. These experiments use YCSB benchmark, and set the number of server nodes to 16 and partitions per transactions to 2.

sets 128 warehouses per server, and, hence a multi-partition can access two co-located partitions (that is on a single server).

Figure 14a represents the scalability of the *Payment* transactions for the three commit protocols. It is evident from this plot that as the number of server nodes increase, the system throughput increases for each commit protocol. However, there



(a) Payment Transaction



(b) NewOrder Transaction

Figure 14: System throughput on varying the number of server nodes, on the TPC-C benchmark. The number of warehouses per server are set to 128.

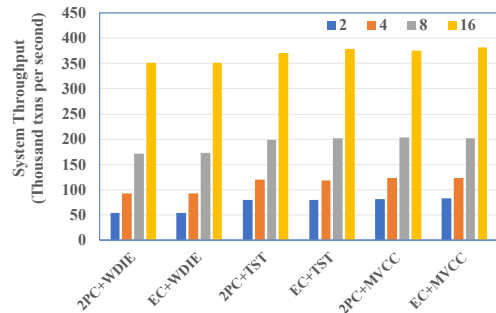


Figure 15: System throughput achieved by three different concurrency control algorithms. For experimentation, we use the TPC-C *Payment* transaction, and vary the number of server nodes to 16. The number of warehouses per server are set to 128. Here WDIE and TST refer to WAIT-DIE and TIMESTAMP, respectively.

is a performance bottleneck in case of 3PC protocol. In case of payment transactions as updates are performed at the home warehouse, which requires exclusive access, so there is an increase in the abort rate for the underlying concurrency control algorithm (in our case NO_WAIT). Now, as 3PC protocol requires an additional phase to commit the transaction, hence there is an increase in the abort rate. Interestingly, the throughput achieved by the EC protocol is approximately equal to the system throughput under 2PC protocol.

Figure 14b depicts the system throughput on executing TPC-C *NewOrder* transactions. The performance bottleneck is reduced for these transactions as there only 10 districts per warehouse, and hence, the commit protocols achieve comparatively higher throughput. Also, as there are only 10% multi-partition transactions, so all the protocols achieve nearly the same performance.

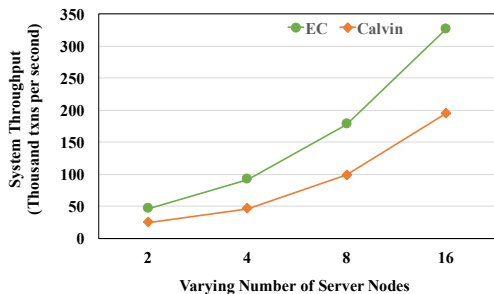


Figure 16: Comparison of throughput achieved by the system executing Calvin versus the system implementing the combination of No-Wait+EC protocol. In this experiment we use the TPC-C *Neworder* transaction, and vary the number of server nodes to 16. The number of warehouses per server are set to 128.

6.7 Concurrency Control

The presence of read/write data conflicts between transactional accesses necessitates the use of concurrency control algorithms by the database management system. The ExpoDB framework implements multiple state-of-the-art concurrency control algorithms. Although, in this work, we use NO_WAIT concurrency control algorithm, but EC protocol can be easily integrated to work alongside other concurrency control algorithms.

Figure 15 measures the system throughput for three different concurrency control algorithms. We use TPC-C *Payment* transactions for these experiments, and increase the number of server nodes upto 16. We also set the number of warehouses per server to 128. We compare the performance of EC protocol against the 2PC protocol, when the underlying concurrency control algorithm is WAIT-DIE [4], TIMESTAMP [4] and MVCC [5]. It is evident from these experiments that the EC protocol is able to achieve as high efficiency as the 2PC protocol, irrespective of the mechanism used for ensuring concurrency control.

We also analyze our commit protocol against an interesting deterministic concurrency control algorithm – *Calvin* [56]. *Calvin* is a deterministic algorithm that requires the prior knowledge of the read/write sets of the transaction before its execution. When the transaction’s read/write sets are not known, at prior, then *Calvin* causes some transactions to execute twice. Interestingly, in the second pass, if some records modify then the transaction is aborted and restarted again. Hence, prior works [28] have shown *Calvin* to perform poorly in such settings. Another strong critic against *Calvin* is that in case of failures, it requires a replica node that executes the same set of operations as the node responding to client query. This implies that Calvin is not suitable under failures for use with partitioned databases. Also, the requirement for replica node, reduces the system throughput.

Figure 16 presents a comparison of NO_WAIT algorithm (employing EC protocol) and Calvin. For this experiment we use the TPC-C *Neworder* transactions, and vary the number of server nodes from 2 to 16. These transactions are required to update the order number in their districts. Hence, the deterministic protocols such as Calvin suffer performance degradation.

7 OPTIMIZATIONS

In earlier sections, we presented a theoretical proof and an evaluation of Easy Commit protocol, which proved its relevance in the space of existing commit protocols. We now discuss some optimizations for the EC protocol.

An optimized version of the EC protocol would allow achieving further gains in comparison to both the 2PC and 3PC protocols. A simple approach is to reduce the number of messages

transmitted in the second phase. In the optimized protocol, each node only forwards messages to those nodes from which it has not received a *Global-Commit* or *Global-Abort* message. Another simple optimization is to ensure early cleanup, that is reduction of implementation enforced wait (refer Section 5.3). To achieve this, each node would maintain a lookup table, where an entry for each transaction is added, on receiving the first *Global-Commit* or *Global-Abort* message. The remaining messages, addressed to the same transaction, would be matched in the table and deleted. We would also need to periodically, flush some of the entries of the table, to reclaim memory. Interestingly, such an optimization would allow implementing a variant of EC protocol that does not require any “implicit” acknowledgments. Note a similar limited variant for 3PC protocol can be constructed where the coordinator does not wait for acknowledgments after sending the *Prepare-to-Commit* messages, and directly transmits *Global-Commit* message to all the cohorts. Our proposed optimized version is comparable to this 3PC variant.

8 RELATED WORK

The literature presents several interesting works [1, 23, 53] that suggest the use of *one phase commit* protocol. These works are strictly targeted at achieving performance, rather than consistency. Clearly, none of these works satisfy the non-blocking requirement, expected of a commit protocol.

Several variants to the 2PC protocol [20, 25, 29, 31, 33, 36, 40, 44, 49] have been proposed that aim at improving its performance. Presumed-commit and presumed-abort [36] work by reducing a single round of message transmission between the coordinator and the participants, when the transaction is to be committed or aborted, respectively. Gray and Reuter [25] present a series of optimizations for enhancing the 2PC protocol such as lazy commit, read-only commit and balancing the load by coordinator transfer. Group commit [20, 40] helps to reduce the commit overhead by committing a batch of transactions together. Samaras et al. [49] design several interesting optimizations to improve the performance of 2PC protocol. They present heuristics to reduce the overhead of logging, network contention and resource conflicts. Compared to all of these works, we present EC protocol, which is not only efficient, but also satisfies the non-blocking property.

Levy et al. [33] present an optimistic 2PC protocol that releases the locks held by a transaction once all the nodes agree to commit. In case a node decides to abort the transaction then to prevent violation of database atomicity, compensating transactions are issued to rollback the changes. Although their approach does not guarantee non-blocking behavior, but we believe the idea of optimistic resource release can be integrated with Easy Commit protocol to achieve further performance.

Boutros and Desai [44] present another variant to 2PC protocol which forces each node to send an additional message in case of a communication failure between the coordinator and the participant. Their approach is only susceptible to the cases where there is a message loss. However, their work does not resolve blocking under site failures and can be integrated with our work to achieve further resilience during message loss.

Haritsa [29] et al. improve the performance of the 2PC protocol, in the context of *real-time* distributed systems. Their protocol permits a conflicting transaction to access the non-committed data. This can lead to cascading aborts, and is not suitable for use with the traditional distributed databases. Our technique, on the other hand, is independent of the underlying concurrency control mechanism, and does not cause any special aborts.

Jiménez-Peris et al. [31] also allow their system to optimistically fetch the uncommitted data, thereby rendering the 2PC performance. However, their protocol is tailored for usage alongside strict two-phase locking, and assumes existence of an additional replica of each process. Our technique is not tailored to any specific concurrency control mechanism, and neither assumes existence of any extra process. Also, we believe these heuristics can be used alongside EC protocol, to render further benefits.

Reddy and Kitsuregawa [58] modify the 3PC protocol by introducing the notion of backup sites. With the help of backup sites they are able to achieve better performance than 3PC, but their approach blocks in case of multiple failures. Easy Commit is non-blocking and does not require any backup sites.

There also have been works [13, 27] that provide better performance bounds than the 3PC protocol if the number of failures are sufficiently less than the participants. Easy Commit does not bound the number of failures, and is nearly as efficient as 2PC.

Gray and Lamport [24] developed an interesting non-blocking version of 2PC protocol using Paxos [32]. Their approach shows that 2PC protocol is a variant of general consensus protocol. However, to ensure non-blocking property they require use of an extra set of acceptor nodes and in the worst case it can be shown that the number of messages transmitted in their approach is $O(n^2)$. Easy Commit is a hybrid between 2PC and 3PC protocol, which is nearly as efficient as former and non-blocking as latter. It also does not require the paxos consensus algorithm, and hence, no additional requirement of acceptor nodes.

9 CONCLUSIONS

We present a novel commit protocol – Easy Commit. Our design of Easy Commit, leverages the best of twin worlds (2PC and 3PC), it is non-blocking (like 3PC) and requires two phases (like 2PC). Easy Commit achieves these goals by ensuring two key observations: (i) first transmit and then commit, and (ii) message redundancy. We present the design of the Easy Commit protocol and prove that it guarantees both safety and liveness. We also present the associated termination protocol and state cases where Easy Commit can perform independent recovery. We perform a detailed evaluation of EC protocol on a 64 node cloud, and show that it is nearly as efficient as the 2PC protocol.

ACKNOWLEDGMENTS

We would like to acknowledge the Microsoft Azure Research Award, which allowed us to conduct extensive experimentation on Microsoft Azure Cloud Services.

REFERENCES

- [1] M. Abdallah, R. Guerraoui, and P. Pucheral. 1998. One-Phase Commit: Does it make Sense? (*ICPADS*).
- [2] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. 2013. Highly Available Transactions: Virtues and Limitations. *Proc. VLDB Endow* 7, 3 (2013), 12.
- [3] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. 2016. Scalable Atomic Visibility with RAMP Transactions. *ACM Trans. Database Syst.* 41, 3 (2016), 45.
- [4] P. A. Bernstein and N. Goodman. 1981. Concurrency Control in Distributed Database Systems. *ACM Comput. Surv.* 13, 2 (1981), 185–221.
- [5] P. A. Bernstein and N. Goodman. 1983. Multiversion Concurrency Control - Theory and Algorithms. *ACM TODS* 8, 4 (1983), 465–483.
- [6] P. A. Bernstein, V. Hadzilacos, and N. Goodman. 1987. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- [7] P. A. Bernstein, V. Hadzilacos, and N. Goodman. 1987. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co.
- [8] J. Bonneau, A. Miller, J. Clark, A. Narayanan, J. A. Kroll, and E. W. Felten. 2015. SoK: Research Perspectives and Challenges for Bitcoin and Cryptocurrencies. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP '15)*. IEEE Computer Society, Washington, DC, USA, 104–121.
- [9] K. Chen, Y. Zhou, and Y. Cao. 2015. Online Data Partitioning in Distributed Database Systems. In *Proceedings of the 18th International Conference on Extending Database Technology, OpenProceeding.org*, 1–12.
- [10] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*. ACM, 143–154.
- [11] Oracle Corporation. Oracle 9i Real Application Clusters concepts Release 2 (9.2), Part Number A96597-01.
- [12] Transaction Processing Performance Council. 2010. TPC Benchmark C (Revision 5.11).
- [13] P. Dutta, R. Guerraoui, and B. Pochon. 2004. Fast Non-blocking Atomic Commit: An Inherent Trade-off. *Inf. Process. Lett.* 91, 4 (2004), 195–200.
- [14] C. Diaconu et al. 2013. Hekaton: SQL Server’s Memory-optimized OLTP Engine. ACM, 1243–1254.
- [15] J. Baker et al. 2011. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*. 223–234.
- [16] J. C. Corbett et al. 2012. Spanner: Google’s Globally-Distributed Database. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX Association, 261–264.
- [17] J. Shute et al. 2013. F1: A Distributed SQL Database That Scales. In *VLDB*.
- [18] R. Kallman et al. 2008. H-store: A High-Performance, Distributed Main Memory Transaction Processing System. *PVLDB* 1 (2008), 1496–1499.
- [19] B. Fung. 2017. The embarrassing reason behind Amazon’s huge cloud computing outage this week. *The Washington Post*, GA, USA.
- [20] Dieter Gawlick and David Kinkade. 1985. Varieties of concurrency control in IMS/VS fast path. 8 (01 1985), 3–10.
- [21] J. Gray. 1978. Notes on Data Base Operating Systems. In *Operating Systems, An Advanced Course*. Springer-Verlag, 393–481.
- [22] J. Gray. 1981. The Transaction Concept: Virtues and Limitations (Invited Paper) (*VLDB*). 144–154.
- [23] J. Gray. 1990. *A comparison of the Byzantine Agreement problem and the Transaction Commit Problem*. Springer New York, 10–17.
- [24] J. Gray and L. Lamport. 2006. Consensus on Transaction Commit. *ACM TODS* 31, 1 (2006), 133–160.
- [25] J. Gray and A. Reuter. 1992. *Transaction Processing: Concepts and Techniques* (1st ed.). Morgan Kaufmann Publishers Inc.
- [26] R. Guerraoui. 1995. *Revisiting the relationship between non-blocking atomic commitment and consensus*. Springer Berlin Heidelberg, 87–100.
- [27] R. Guerraoui, M. Larrea, and A. Schiper. 1996. Reducing the Cost for Non-blocking in Atomic Commitment. IEEE.
- [28] R. Harding, D. Van Aken, A. Pavlo, and M. Stonebraker. 2017. An Evaluation of Distributed Concurrency Control. *Proc. VLDB Endow* 10, 5 (2017), 553–564.
- [29] Jayant R. Haritsa, Krithi Ramamritham, and Ramesh Gupta. 2000. The PROMPT Real-Time Commit Protocol. *IEEE TPDS* 11, 2 (2000), 160–181.
- [30] M. P. Herlihy and J. M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM TOPLAS* 12, 3 (1990).
- [31] Ricardo Jiménez-Peris, Marta Patiño Martínez, Gustavo Alonso, and Sergio Arévalo. 2001. A Low-Latency Non-blocking Commit Service (*DISC’01*). Springer-Verlag.
- [32] L. Lamport. 1998. The Part-time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (1998), 133–169.
- [33] E. Levy, H. F. Korth, and A. Silberschatz. An Optimistic Commit Protocol for Distributed Transaction Management (*ACM SIGMOD*). ACM, 88–97.
- [34] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. 2013. Stronger Semantics for Low-latency Geo-replicated Storage (*NSDI*). USENIX Association, 313–328.
- [35] MemSQL. <http://www.memsql.com>.
- [36] C. Mohan, B. Lindsay, and R. Obermarck. 1986. Transaction Management in the R* Distributed Database Management System. *ACM TODS* 11, 4 (1986).
- [37] NuoDB. <http://www.nuodb.com>.
- [38] S. A. O’Brien. Facebook, Instagram experience outages Saturday. CNN, GA, USA.
- [39] M. T. Ozsu and P. Valduriez. 2011. *Principles of Distributed Database Systems* (3rd ed.). Springer-Verlag, 428–453 pages.
- [40] T. Park and H. Y. Yeom. 1991. A Distributed Group Commit Protocol for Distributed Database Systems (*ICPADS*).
- [41] S. Patterson, A. J. Elmore, F. Nawab, D. Agrawal, and A. El Abbadi. 2012. Serializability, Not Serial: Concurrency Control and Availability in Multi-datacenter Datastores. *Proc. VLDB Endow* 5, 11 (2012).
- [42] A. Pavlo, C. Curino, and S. Zdonik. 2012. Skew-aware Automatic Database Partitioning in Shared-nothing, Parallel OLTP Systems (*SIGMOD ’12*). ACM, 61–72.
- [43] M. Pilkington. 2015. Blockchain Technology: Principles and Applications. In *Research Handbook on Digital Transformations*. SSRN.
- [44] B. S. Boutros and B. C. Desai. A two-phase commit protocol and its performance (*DEXA*). IEEE, 100–105.
- [45] M. Sadoghi. 2017. ExpoDB: An Exploratory Data Science Platform. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017*.
- [46] M. Sadoghi, S. Bhattacharjee, B. Bhattacharjee, and M. Canim. 2018. L-Store: A Real-time OLTP and OLAP System (*EDBT*). OpenProceeding.org.
- [47] M. Sadoghi, M. Canim, B. Bhattacharjee, F. Nagel, and K. A. Ross. 2014. Reducing Database Locking Contention Through Multi-version Concurrency. *Proc. VLDB Endow* 7, 13 (Aug. 2014), 1331–1342.
- [48] M. Sadoghi, K. A. Ross, M. Canim, and B. Bhattacharjee. 2013. Making Updates disk-I/O Friendly Using SSDs. *Proc. VLDB Endow* 6, 11 (Aug. 2013), 997–1008.
- [49] G. Samarak, K. Britton, A. Citron, and C. Mohan. 1995. Two-phase commit optimizations in a commercial distributed environment. *Distributed and Parallel Databases* 3, 4 (1995), 325–360.
- [50] D. Skeen. 1981. Nonblocking Commit Protocols (*SIGMOD*). ACM, 133–142.
- [51] D. Skeen. 1982. *A Quorum-Based Commit Protocol*. Technical Report.
- [52] D. Skeen and M. Stonebraker. 1983. A Formal Model of Crash Recovery in a Distributed System. *IEEE Trans. Softw. Eng.* 9, 3 (1983), 219–228.
- [53] J.W. Stamos and F. Cristian. 1990. A Low-Cost Atomic Commit Protocol. In *Proceedings of the 9th Symposium on Reliable Distributed Systems*. IEEE, 10–17.
- [54] M. Stonebraker. 1986. The Case for Shared Nothing. *Database Engineering* 9 (1986), 4–9.
- [55] A. Sulleyman. Twitter Down: Social Media App And Website Not Working. The Independent, UK.
- [56] A. et al. Thomson. 2012. Calvin: Fast Distributed Transactions for Partitioned Database Systems (*SIGMOD*).
- [57] VoltDB. <https://www.voltdb.com/>.
- [58] P. K. Reddy and M. Kitsuregawa. 1998. Reducing the Blocking in Two-Phase Commit Protocol Employing Backup Sites. In *COOPIS’98*. IEEE, 406–416.

Beyond Frequencies: Graph Pattern Mining in Multi-weighted Graphs

Giulia Preti
University of Trento
gp@disi.unitn.eu

Davide Mottin
Hasso Plattner Institute
davide.mottin@hpi.de

Matteo Lissandrini
University of Trento
ml@disi.unitn.eu

Yannis Velegrakis
University of Trento
velgias@disi.unitn.eu

ABSTRACT

Graph pattern mining aims at identifying structures that appear frequently in large graphs, under the assumption that frequency signifies importance. Several measures of frequency have been proposed that respect the *apriori* property, essential for an efficient search of the patterns. This property states that the number of appearances of a pattern in a graph cannot be larger than the frequency of any of its sub-patterns. In real life, there are many graphs with weights on nodes and/or edges. For these graphs, it is fair that the importance (score) of a pattern is determined not only by the number of its appearances, but also by the weights on the nodes/edges of those appearances. Scoring functions based on the weights do not generally satisfy the *apriori* property, thus forcing many approaches to employ other, less efficient, pruning strategies to speed up the computation. The problem becomes even more challenging in the case of multiple weighting functions that assign different weights to the same nodes/edges. In this work, we provide efficient and effective techniques for mining patterns in multi-weight graphs. We devise both an exact and an approximate solution. The first is characterized by intelligent storage and computation of the pattern scores, while the second is based on the aggregation of similar weighting functions to allow scalability and avoid redundant computations. Both methods adopt a scoring function that respects the *apriori* property, and thus they can rely on effective pruning strategies. Extensive experiments under different parameter settings prove that the presence of edge weights and the choice of scoring function affect the patterns mined, and hence the quality of the results returned to the user. Finally, experiments on datasets of different sizes and increasing numbers of weighting functions show that, even when the performance of the exact algorithm degrades, the approximate algorithm performs well and with quite good quality.

1 INTRODUCTION

Pattern mining in large graphs has attracted considerable attention, since it finds applications in many real world scenarios like fraud detection [31], biological structures identification [16], anticipation of user intention [33], graph similarity search [20], traffic control [21], and query optimization [42]. It has been studied for graph collections [41], for attributed [35], probabilistic [26], or even generic large graphs [10]. The goal is to identify patterns that occur frequently, given that frequency indicates importance. An interesting property regarding frequency is that a pattern

cannot be more frequent than any of its sub-patterns, known as the *apriori* property. This property enables efficient implementations [43], as it ensures that the frequency of a pattern decreases monotonically as the pattern grows in size, thus allowing the mining process to start from small patterns and extend to larger ones only when the frequency of the pattern is above a certain frequency threshold.

In graph databases the frequency of a pattern has been effectively computed as the number of distinct graphs containing an appearance of the pattern. However, the same implementation cannot be used in single large graphs, as each pattern would have frequency either equal to 0 or to 1. Furthermore, if we simply define the frequency as the number of distinct occurrences of the pattern, we may break the *apriori* property [38]. In fact, this implementation counts every overlap that may occur among the occurrences, hence assigning larger frequencies to larger patterns, causing an unwanted (and unjustified) skew in the value of importance for some patterns. For this reason, alternative metrics have been considered in the literature [6, 11, 38], with the more prevalent one being the MNI support, as it enjoys high effectiveness [10].

Many real world scenarios are naturally modeled through weighted graphs, and in these cases, the importance of a pattern should be determined not only by the frequency, but also by the weights of its appearances. Examples include the discovery of metabolic pathways in genomic networks [23], where weights indicate strength between genomes [8], the identification of topics of interest in large knowledge graphs [29], where weights quantify the degree a piece of data is qualified as an answer to a user [39], or the detection of common problematic cases in computer networks, where weights indicate congestion [5]. Unfortunately, weighted graphs do not possess the *apriori* property, since the weights of the extra edges/nodes of a larger pattern may offset its lower frequency. As a consequence, some works that considered weighted graphs for pattern mining proposed solutions that are less efficient than those based on the *apriori* property [43].

A requirement in modern applications is to offer personalized products and services rather than generic preferences [34]. Such generic preferences suit the user on average but fail to deliver the right answer for each specific user. The same argument holds for graph patterns. For instance, social network systems record user interactions [22] and activities [4] and build graphs by modeling the relationships among users and web content to find frequent patterns of interactions [30]. Advertisers subsequently exploit such patterns to target the right customer for a certain product. Some patterns of interactions may be more important than others to an advertiser depending on the product or the specific business model; in such case, multiple weights are valuable. Other

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

examples include online retailers like Amazon, which build large graphs on product co-purchases and then exploit the discovered patterns to recommend future offers [36]. Frequency, number of items, recency of the purchase, as well as the company’s business intentions affect the importance of some co-purchases with respect to others [34]. Such examples highlight the need for a solution that accounts for the individual preferences expressed as a multi-weighted graph, as opposed to “one size fits all” solutions. However, the straightforward approach to multi-weighted pattern mining that runs the mining algorithm on each weighted graph separately, is clearly impractical due to the graph size and the large number of users.

In this paper, we propose a novel approach to mine patterns in weighted graphs that goes beyond frequencies, yet has performance not significantly different from the pattern mining in unweighted graphs. We achieve this by defining a family of scoring functions that are based on the MNI score [6], a metric that is widely used in graph mining due to its characteristic of respecting the apriori property, while being efficient to compute. The solution we have devised is modeled as a constraint satisfaction problem (CSP), as proposed also for unweighted pattern mining [10], and implements the pattern growth approach, as introduced by gSpan [41]. Furthermore, we extend the idea above for the case of graphs with multiple weights on their edges/nodes. To avoid running the algorithm one time for each different weighting function, we compute all the scores of each pattern at the moment we are visiting it, and keep the patterns that return a high score with respect to at least one weighting function.

In particular we make the following contributions:

- (a) We extend the task of pattern mining in weighted large graphs for a novel family of scoring functions based on the MNI support [6] (Section 2).
- (b) We introduce and formally define the problem of pattern mining in multi-weight graphs with different weighting functions.
- (c) We devise two efficient and effective techniques for solving the pattern mining problem on weighted graphs (Section 3). The first one is an exact solution, called **RESUM**, that is less time and space consuming than the (naive) brute force method. It avoids redundant revisits of the graph, by aggregating and performing once multiple computations on the same parts of the graph, and storing the relevant patterns in a compact way. The second one is a conservative approximate solution, called **RESUM approximate**, that reduces the number of weighting functions to consider, by aggregating those having a high probability to generate similar results (Section 4) into a single representative function. In addition, we show that this method introduces only few false positives, while running considerably faster than the exact approach.
- (d) We study four different scoring functions (all based on the MNI support) for devising the score of a pattern in an efficient way (Section 5).
- (e) We evaluate our approach with an extensive set of experiments and discuss our findings (Section 7).

2 PROBLEM DEFINITION

We assume the existence of a countable set of labels Σ that includes the special symbol \perp , and a set $\mathcal{I}_0^1 = [0, 1] \cup \{\perp\}$ of weights. The symbol \perp in Σ and \mathcal{I}_0^1 is used to denote *no label* and *no weight*, respectively. A weighted graph is a structure that consists of a set of nodes, a set of edges between them, an assignment of labels

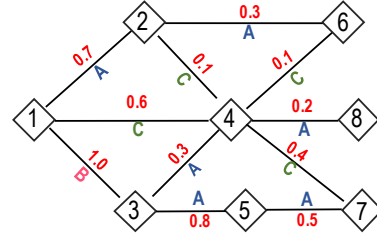


Figure 1: Example of an edge-labeled, weighted graph.

to all the nodes and edges, and an assignment of weights to all the edges.

Definition 2.1. A **weighted labeled graph**, or simply a **graph**, is a tuple $\langle V, E, \ell, \omega \rangle$ where V is a set of vertices, $E \subseteq V \times V$ is a set of edges, $\ell : E \cup V \rightarrow \Sigma$ is a labeling function, and $\omega : E \rightarrow \mathcal{I}_0^1$ is a weighting function. The symbol \mathcal{G} is used to denote the set of all the possible graphs.

Note that we assume weights on edges only, mainly for presentation purposes. Weights on nodes can also be considered with no need for any major modification. A graph $S : \langle V_S, E_S, \ell, \omega \rangle$ is said to be a **subgraph** of another graph $G : \langle V_G, E_G, \ell, \omega \rangle$, denoted as $S \sqsubseteq G$, if $V_S \subseteq V_G$ and $E_S \subseteq E_G$. Note that the two graphs have the same labeling and weighting function.

To express the fact that two graphs have the same topological structure, we use the notion of *isomorphism*, which is a bijective mapping between the nodes of the two graphs such that the edges between the nodes, alongside their labels, are preserved through the mapping.

Definition 2.2. A graph $G : \langle V, E, \ell, \omega \rangle$ is **isomorphic** to a graph $G' : \langle V', E', \ell', \omega' \rangle$, denoted as $G \simeq G'$, if there exists a bijective function $\phi : V \rightarrow V'$ such that: $\forall \langle u, v \rangle \in E : \langle \phi(u), \phi(v) \rangle \in E'$ and $\ell(\langle u, v \rangle) = \ell'(\langle \phi(u), \phi(v) \rangle)$.

A graph G may have multiple isomorphic graphs. To collectively represent those graphs, we introduce the concept of *pattern*. Intuitively, a pattern is a graph with no weights, serving as a representative of a set of isomorphic graphs and describing their common structure.

Definition 2.3. A **pattern** is a graph $\langle V, E, \ell, \omega \rangle$, such that $\forall e \in E : \omega(e) = \perp$. The symbol \mathcal{P} denotes the set of all possible patterns. Given a graph G , and a pattern P , the *support set* of the pattern P is the set $S_G(P) = \{g | g \sqsubseteq G \wedge g \simeq P \wedge P \in \mathcal{P}\}$. Each element in $S_G(P)$ is referred to as an *appearance* (or *matching*) of P in G .

By definition, the support set of P is the set of all the subgraphs of G that are isomorphic to P . By abuse of notation we write $P \sqsubseteq G$ and call the pattern P a subgraph of G if its support set is non-empty. Then, We denote by ϕ_g^P the bijection that maps an isomorphic subgraph g of G to the pattern P .

Given a **scoring function** $f : \mathcal{P} \times \mathcal{G} \rightarrow \mathbb{R}$, we will refer to the value $f(P, G)$ as the **score** of P in G . Graph pattern mining is the task that aims at identifying those patterns that have score higher than a threshold τ , or the k patterns with the highest score [10]. A natural scoring function is the one that returns the cardinality of $S_G(P)$, i.e., the number of appearances of the pattern P in the graph G , and the patterns identified by this function are called *frequent patterns*. Nevertheless, it has been shown that this simple function violates the a-priori property, due to the presence of overlapping isomorphisms in G [6]. As an example, the frequency of the pattern $P_1 : [v_1] - B - [v_2] - A - [v_3]$ in the graph in Figure 1

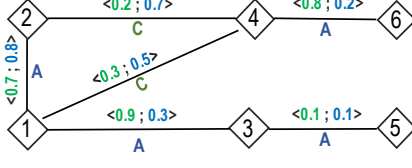


Figure 2: Graph with two weights $\langle \omega_1, \omega_2 \rangle$ on each edge.

is 3, while the frequency of its sub-pattern $P_2 : [v_1] - B - [v_2]$ is 1. For this reason a number of works have investigated alternative scoring functions [11, 15, 25, 38]. Among them, the MNI support is highly effective and efficient to compute [6].

Definition 2.4. Given a graph $G : \langle V, E, \ell, \omega \rangle$, the **MNI support** of a pattern $P : \langle V_P, E_P, \ell_P, \omega_P \rangle$ in G is the number $MNI(P, G) = \min_{v' \in V_P} |\mathcal{N}(G, v')|$ where $\mathcal{N}(G, v') = \{v \mid v \in V \wedge \exists g \in S_G(P) \text{ such that } \phi_g^P(v) = v'\}$.

Intuitively, the set $\mathcal{N}(G, v')$ contains all the nodes of G that are mapped to the pattern node v' by some isomorphism ϕ_g^P from g to P . Then, the MNI support is the minimum cardinality of this set across all the nodes of the pattern P . We can define similar sets also for the pattern edges, i.e., for each $e' \in E_P$, the set $\mathcal{E}(G, e') = \{e \mid e \in E \wedge \exists g \in S_G(P) \text{ such that } \phi_g^P(e) = e'\}$ contains all the edges of G that are mapped to the pattern edge e' by some isomorphism ϕ_g^P . Consider, for instance, the graph G in Figure 1 and the pattern $P : [v_1] - B - [v_2] - A - [v_3]$. Since $\mathcal{N}(G, v_1) = \{1, 3\}$, $\mathcal{N}(G, v_2) = \{1, 3\}$, and $\mathcal{N}(G, v_3) = \{2, 4, 5\}$, the MNI support of P is 2. On the other hand, the number of appearances of P in G is 3: $S_G(P) = \{[3] - B - [1] - A - [2], [1] - B - [3] - A - [4], [1] - B - [3] - A - [5]\}$.

In the presence of weights on edges, the score of a pattern cannot be based only on the frequency, but should strike a balance between frequency and weights, allowing also the weights to play a role in assessing the relevance of the pattern. Thus, there is a need for a different scoring function that looks beyond the structure of the subgraphs, by considering the importance of their edges as well. In this case, we talk about *weighted frequent patterns*, or *relevant patterns*. This alternative scoring function, however, has to be carefully selected to satisfy the apriori property [43].

Furthermore, if there are multiple weighting functions, i.e., several functions that assign weights on the graph edges/nodes, then the pattern mining task must be carried out for each individual function. An example of graph with multiple weights on the edges is illustrated in Figure 2. Such situation leads to the following specification of the mining task.

Pattern Mining in Multi-Weighted Graphs. Given a threshold τ , a scoring function f and a graph $G : \langle V, E, \ell, W \rangle$, where W is a finite set of weighting functions, we must discover, $\forall \omega_i \in W$, the set of patterns $R_i = \{P \mid G' = \langle V, E, \ell, \omega \rangle \wedge f(P, G') \geq \tau\}$.

3 SCORE-BASED PATTERN MINING

Our solution to the pattern mining on weighted graphs problem consists of two steps. The first step is the identification of the frequent patterns and the elimination of the appearances that do not satisfy the constraints on the weights imposed by the scoring function used. In the second step, a score is computed for each pattern (and for each weighting function in the case of multi-weighted graphs) in terms of the appearances in its support set that were selected in the first step.

Algorithm 1 RELEVANTPATTERNMINING

Input: Graph $G : \langle V, E, \ell, \omega \rangle$, min score τ

Output: Set of relevant patterns R

```

1:  $R \leftarrow \text{RELEVANTEDGES}(G)$ 
2:  $fE \leftarrow \text{FREQUENTEDGES}(G)$ 
3: while  $fE \neq \emptyset$  do
4:    $e \leftarrow fE.\text{pop}$ 
5:    $R \leftarrow R \cup \text{PATTERNEXTENSION}(G, e, \tau, fE \cup \{e\})$ 
6: return  $R$ 

7: function  $\text{PATTERNEXTENSION}(G, g, \tau, fE)$ 
8:    $Cand \leftarrow \emptyset; \mathfrak{S} \leftarrow \emptyset$ 
9:   for each  $e \in fE$  do
10:     $Cand \leftarrow Cand \cup \{g \diamond e\}$ 
11:   for each  $c \in Cand$  do
12:     $(score, sup) \leftarrow \text{EXAMINEPATTERN}(G, c)$ 
13:    if  $sup \geq \tau$  then
14:       $\mathfrak{S} \leftarrow \mathfrak{S} \cup \text{PATTERNEXTENSION}(G, c, \tau, fE)$ 
15:    if  $score \geq \tau$  then
16:       $\mathfrak{S} \leftarrow \mathfrak{S} \cup \{c\}$ 
17:   return  $\mathfrak{S}$ 

```

3.1 Assessing the relevance of a pattern

A scoring function can have many different properties, which may be desirable for certain applications but not for others. As a consequence there may be no scoring function that is consistently better than others in all the applications. Therefore, in this work we do not advocate a single one-size-fits-all scoring function, but we propose a framework that can accommodate a wide range of functions.

Assuming w.l.o.g. that larger weights signify higher importance, the function f must satisfy the following key properties:

P1: the score $f(P, G)$ monotonically increases with the weights of its appearances.

P2: the score $f(P, G)$ monotonically increases with the number of appearances with large weights.

P3: f is *MNI-compatible*, i.e., $f(P, G) \geq \tau \implies MNI(P, G) \geq \tau$.

Properties **P1** and **P2** are a natural consequence of our assumption on the importance of the weights, while **P3** ensures the time practicality of the solution.

3.2 Mining weighted graphs

Finding the frequent patterns on weighted graphs requires the computation of the frequency and the score of each pattern. To this end, we propose RESUM, an efficient and effective general algorithm for *any* MNI-compatible score that exploits the pruning power of the anti-monotonicity property of the MNI support.

We model the frequent subgraph mining as a *constraint satisfaction problem* (CSP) [9]. An instance of the CSP problem is a tuple (X, D, C) where X is a set of variables, D is a set of domains corresponding to the variables in X , and C is a set of constraints between the variables in X . A solution for an instance of CSP is an assignment from the candidates in D to the variables in X that satisfies all the constraints in C . The matching problem for a pattern $P \sqsubseteq G$ is then translated into $CSP(P) = (X_P, D_P, C_P)$, so that any solution for $CSP(P)$ corresponds to a subgraph g isomorphic to P .

Specifically, each node $v \in V_P$ is mapped to a variable $x_v \in X_P$, each domain $D_v \in D_P$ is a subset of V containing all the graph

Algorithm 2 EXAMINEPATTERN

Input: Graph $G:(V, E, \ell, \omega)$, pattern P , score threshold τ **Output:** Score and MNI support of P

```
1: for each  $v \in V_P$  do
2:    $sup_v \leftarrow \emptyset$ 
3:    $D_v \leftarrow \{v' \in V | \ell(v') = \ell(v)\}$ 
4:    $\mathcal{A} \leftarrow$  automorphisms of  $P$ 
5:   STRUCTURALCONSISTENCY( $\{D_v | v \in V_P\}, P$ )
6:   for each  $v \in V_P$  do
7:     if  $\exists w = \mathcal{A}(v)$  s.t.  $D_w$  already computed then
8:        $D_v \leftarrow D_w$ 
9:       continue
10:    STRUCTURALCONSISTENCY( $\{D_v | v \in V_P\}, P$ )
11:    if  $\exists D_u$  s.t.  $|D_u| < \tau$  then return  $(-1, -1)$ 
12:    for each  $n \in D_v$  do
13:      search for  $g$  s.t.  $g \simeq P \wedge n \in V_g \wedge n \mapsto v$ 
14:      if  $g \neq Nil$  then
15:        Valid  $\leftarrow$  ISVALID( $g, \omega$ )
16:        for each  $n' \in V_g, v' \in V_P$  s.t.  $n' \mapsto v'$  do
17:          mark  $n'$  in  $D_{v'}$ 
18:          if Valid then
19:             $sup_{v'} \leftarrow sup_{v'} \cup \{n'\}$ 
20:        else
21:          remove  $n$  from  $D_v$ 
22:   $score \leftarrow$  RELEVANCESCORE( $\{sup_v | v \in V_P\}$ )
23:   $mni \leftarrow \min_{v \in V_P} |D_v|$ 
24:  return  $(score, mni)$ 
```

nodes isomorphic to v , and C includes consistency constraints that enforce a topology isomorphic to that of P [28]. Then, for each candidate node $n \in D_v$ we search for a valid assignment that maps n to v . If no assignment is found, n is removed from the domain D_v and the topology constraints are checked again until no invalid candidate is found in the other domains. At the end of the process, the number of elements in the smallest domain, i.e., $\text{argmin}_{D_v \in D_P} |D_v|$, corresponds to the MNI support of P , as defined in Definition 2.4. Therefore, given a score threshold τ , P is frequent if each variable in X_P has at least τ distinct valid assignments. This means that if the size of some domain D_u is lower than τ , P cannot be frequent. Notice that in general not all the matching subgraphs of a pattern satisfy the constraints on the weights forced by the scoring function used, and thus we must additionally check each of them to determine if it contributes to the score of the pattern. The aggregated score is then computed considering only the matches not discarded.

Algorithm 1 outlines the RESUM framework. First, the relevant and the frequent edges are found (Lines 1-2). Then, each subgraph is recursively extended following the pattern-growth approach introduced by gSpan [41] (Line 5), until no other extension is possible. Each extension is a candidate relevant pattern, whose MNI support is computed alongside its *score* by the EXAMINEPATTERN procedure (Algorithm 2). This procedure first initializes the candidate domain D_v of each pattern node $v \in V_P$ with all the nodes in G with the same label as v (Lines 1-3), and the support set sup_v of each node $v \in V_P$ with the empty set. Then, the algorithm computes the automorphisms of the pattern (Line 4). Automorphisms are isomorphisms of a graph to itself and can be used to compute the valid assignments more efficiently (Lines 7-8), since each assignment valid for a pattern node v is valid for each automorphic node w too. Finally the

algorithm iterates over each candidate node $n \in D_v$ to determine if it belongs to some subgraph g isomorphic to P (Lines 12-13). As soon as such subgraph is found, all the domains are updated (Lines 16-17) and the subgraph is checked for validity (Line 15). In particular, the ISVALID procedure compares the edge weights in g against the constraints specified by the scoring function f , and if g satisfies the condition, the nodes of the subgraph are stored in the corresponding support sets (Line 19). These nodes will contribute to the relevance *score* of P .

On the other hand, if n does not participate in any isomorphism, it is removed from D_v . As a consequence, in the subsequent iteration, structural constraints like the minimum degree of a node mapped to a $v \in V_P$ are enforced, to remove candidates that can no longer participate to any isomorphism of P (Line 10). The algorithm terminates either when all the pattern nodes have been examined, or when the size of some domain becomes lower than τ , as in this case P can be neither relevant nor frequent (Line 11). In the first case, instead, the MNI support and the relevance *score* of P are calculated and returned. We refer to Section 5 for a discussion about suitable scoring functions that can be implemented in Procedure ISVALID.

Finally in Lines 13-17 of Algorithm 1, all the frequent patterns are further extended, while all the relevant patterns are included in the final set of relevant patterns R . Since we enforce the use of MNI-compatible scoring functions, the MNI support of P is an upper-bound of its *score*, and thus the pruning strategy ensures that all the relevant patterns are returned.

Complexity. Even though the computation of the automorphisms ($\mathcal{O}(|V_P|^{V_P})$) and the pruning strategy improve the expected performance of the algorithm, in the worst case it takes $C = \mathcal{O}(2^{|V|^2} |V|^{V_P})$ time, which is exponential in the number of nodes and the size of the patterns. In particular, $\mathcal{O}(2^{|V|^2})$ is the time required to compute all the patterns in G , and $\mathcal{O}(|V|^{V_P})$ is that needed to find all the isomorphisms of a pattern P .

3.3 Mining in multi-weighted graphs

In the case of multiple edge weights assigned by m weighting functions $W = \{\omega_1, \dots, \omega_m\}$, the naive approach for solving the Pattern Mining in Multi-Weighted Graphs problem runs Algorithm 1 $|W|$ times, once for each function. Evidently, this approach becomes impractical for large m , as the process of mining the patterns is computationally intense. In fact, this process would take $\mathcal{O}(C^m)$ to terminate.

The naive approach recomputes the same patterns multiple times, incurring in a significant time overhead that can be avoided by running the algorithm only once and keeping track of the relevant patterns for each weighting function. This strategy replaces Line 12 in Algorithm 1 with Algorithm 3, which searches for the isomorphisms of the pattern P , while checking their validity with respect to each $\omega_i \in W$, at the same time. Similarly to the single weight case, we initialize each candidate domain and all the support sets for each weighting function (Lines 1-4). When an isomorphic subgraph is found, procedure ISVALID checks *in parallel* each set of edge weights against the constraints set by the scoring function and stores the results in the auxiliary array VAL . If the weights assigned by ω_i satisfy the constraints, the nodes of the subgraph are stored in the corresponding sets $SUP_v[i]$ (Line 21).

Finally, all the scores of the candidate pattern c are evaluated in Line 16 of Algorithm 1, and c is added to the final set R only if at least one of its scores is larger than τ . As a further optimization,

Algorithm 3 EXAMINESUBGRAPHMULTI

Input: Graph $G: \langle V, E, \ell, W \rangle$, pattern P , score threshold τ **Output:** Scores and MNI support of P

```
1: for each  $v \in V_P$  do
2:    $D_v \leftarrow \{v' \in V \mid \ell(v') = \ell(v)\}$ 
3:   for each  $i \in 1, \dots, |W|$  do
4:      $SUP_v[i] \leftarrow \emptyset$ 
5:  $\mathcal{A} \leftarrow$  automorphisms of  $P$ 
6: STRUCTURALCONSISTENCY( $\{D_v \mid v \in V_P\}, P$ )
7: for each  $v \in V_P$  do
8:   if  $\exists w = \mathcal{A}(v)$  s.t.  $D_w$  already computed then
9:      $D_v \leftarrow D_w$ 
10:    continue
11: STRUCTURALCONSISTENCY( $\{D_v \mid v \in V_P\}, P$ )
12: if  $\exists D_u$  s.t.  $|D_u| < \tau$  then return ( $\{-1, \dots, -1\}, -1$ )
13: for each  $n \in D_v$  do
14:   search for  $g$  s.t.  $g \simeq P \wedge n \in V_g \wedge n \mapsto v$ 
15:   if  $g \neq Nil$  then
16:      $VAL \leftarrow$  ISVALID( $g, W$ )
17:     for each  $n' \in V_g, v' \in V_P$  s.t.  $n' \mapsto v'$  do
18:       mark  $n'$  in  $D_{v'}$ 
19:       for each  $i \in 1, \dots, |W|$  do
20:         if  $VAL[i]$  then
21:            $SUP_{v'}[i] \leftarrow SUP_{v'}[i] \cup \{n'\}$ 
22:       else
23:         remove  $n$  from  $D_v$ 
24:  $\mathcal{S} \leftarrow$  RELEVANCESCORES( $\{SUP_v \mid v \in V_P\}$ )
25:  $mni \leftarrow \min_{v \in V_P} |D_v|$ 
26: return ( $\mathcal{S}, mni$ )
```

instead of storing in memory the sets of relevant patterns for each function ω_i , we maintain a binary vector of size m for each relevant pattern P , where position i is set to 1 if P is relevant for ω_i .

4 APPROXIMATE ALGORITHM

The exact algorithm introduced in Section 3 incurs a significant memory overhead when the number of weighting functions is in the order of thousands, which, for example, is the case for recommender systems for big retailers (e.g., Amazon). For such applications, we devise a more conservative approximate solution, called *ReSUM approximate*, that significantly reduces the memory consumption by taking advantage of the similarities between the weighting functions $\omega_1, \dots, \omega_m \in W$.

The *ReSUM approximate* algorithm first generates $k \ll m$ representative functions ω_j^* , by clustering and aggregating the original functions ω_i . Then, it runs Algorithm 3 to compute k sets of relevant patterns R_1^*, \dots, R_k^* , which are used to build m approximate sets of relevant patterns A_1, \dots, A_m , returned in place of the exact sets R_1, \dots, R_m . Clearly, the quality of the approximate result depends on the way the representative functions are generated. With our implementation, we aim at returning a set A_j for each ω_i that resembles the exact set R_i as much as possible.

4.1 Generation of the representative functions

The generation of the representative functions is shown in Algorithm 4 and consists of three steps. First, each weighting function

Algorithm 4 GENERATE REPRESENTATIVE FUNCTIONS

Input: Graph $G : \langle V, E, \ell, W \rangle$, number of buckets b , number of clusters k **Output:** Set of representative functions W^*

```
1:  $\mathcal{F} \leftarrow$  CREATEFEATUREVECTORS( $E, W, b$ )
2:  $C \leftarrow$  COMPUTECLUSTERING( $\mathcal{F}, k$ )
3:  $W^* \leftarrow$  GENERATEMAXWEIGHTVECTORS( $C, W$ )
4: return  $W^*$ 
```

Algorithm 5 CREATEBUCKETFEATUREVECTORS

```
1: function CREATEBUCKETFEATUREVECTORS( $E, W, b$ )
2:   for each  $l \in \Sigma_E$  do
3:      $BucketList_l \leftarrow$  COMPUTEBUCKETLIMITS( $E_l, W, b$ )
4:     for each  $\omega_i \in W$  do
5:        $r_i^l \leftarrow$  FILLBUCKETS( $E_l, \omega_i, BucketList_l$ )
6:   for each  $\omega_i \in W$  do
7:      $r_i \leftarrow$  CONCAT( $\{r_i^l \mid l \in \Sigma_E\}$ )
8:   return  $\{r_1, \dots, r_{|W|}\}$ 
```

$\omega_i \in W$ is transformed into a feature vector (Line 1). Secondly, the weighting functions are clustered into k groups of similar functions (Line 2). Thirdly, the set of k representative functions $W^* = \{\omega_1^*, \dots, \omega_k^*\}$ is returned (Lines 3-4).

Creation of the feature vectors.

In the first step, we construct a feature vector r_i for each ω_i , which is used in the second step to determine the similarities between the functions. Since our final goal is to assign a set of patterns A_i to each ω_i that is as close as possible to the exact set R_i , a straightforward choice is to use the edge weights as features. We call this approach *full-vector strategy*. According to this strategy, Procedure 1 decides an ordering of the graph edges and creates m vectors r_1, \dots, r_m of size $|E|$, where $r_i[x]$ is the weight assigned by ω_i to the edge in the x^{th} position.

Although similar edge weights lead, with high probability, to similar sets of relevant patterns, the effectiveness and the efficacy of the full-vector strategy decrease as the size of the graph increases. In fact, the high dimensionality of the vectors complicates the detection of functions with similar properties, as a consequence of the curse of dimensionality phenomenon [37].

Thus, we propose also a more efficient approach called *bucket-based strategy*, which overcomes the problem of high dimensionality by considering the edge labels in place of the graph edges, as features to build the vectors. The underlying idea is that, in real scenarios, a preference for an edge is highly correlated with the preference for the label of that edge. This strategy is implemented in Procedure CREATEBUCKETFEATUREVECTORS (Algorithm 5), which takes the set of weighting functions W and the number of buckets b , and generates a set of feature vectors r_1, \dots, r_m each of size $|\Sigma_E| \cdot b$, where Σ_E indicates the set of distinct edge labels. In particular, each vector r_i is the concatenation of $|\Sigma_E|$ summaries of the edge-weights of ω_i , one for each edge label, and b is the resolution of each summary.

The summary for a label l is obtained by splitting the range of admissible weights $[0, 1]$ into b of sub-ranges (buckets) (Line 3), e.g., $[0, x_1)$, $[x_1, x_2)$, and $[x_2, 1.0]$ for $b = 3$. Then Procedure FILLBUCKETS (Line 5) counts, for each sub-range, how many times the function ω_i assigns a weight within that sub-range to an edge with label l . Note that, in the degenerate case of $b = 1$, the vector

r_i simply keeps, for each label, the number of edges with that label and whose weight is greater than 0.

The *bucketization* of a label l is performed by Procedure COMPUTEBUCKETLIMITS (Line 3) following the equi-depth paradigm [13], which assigns the input values to buckets, while trying to balance the number of elements in each bucket. Thus, we consider all the weights assigned by all the weighting functions to edges with label l , and split the range $[0, 1]$ into b depth-balanced intervals.

For example, given $b = 2$, the label ordering $A|C$, and the two weighting functions ω_1 and ω_2 in Figure 1, we obtain the vectors $r_1 = [1, 3, 2, 0]$ and $r_2 = [3, 1, 0, 2]$. As such, the buckets of A are the ranges of values $[0, 0.3]$ and $[0.3, 1]$, and those of C the ranges $[0, 0.5]$ and $[0.5, 1]$.

Note that the bucket-based strategy allows us to decide the size of the feature vectors a priori and tune the parameter b to improve the accuracy of the clustering.

Identification of similar functions. Procedure COMPUTECLUSTERING (Algorithm 4, Line 2) implements the Lloyd’s clustering algorithm [27], which identifies groups of similar ω_i by comparing the feature vectors $r_1, \dots, r_m \in \mathcal{F}$ using the cosine similarity.

The algorithm can be initialized either providing k random seeds among all the vectors in \mathcal{F} , or by selecting the k most diverse feature vectors. Note that finding the most diverse vectors may increase the running time of the algorithm, but this strategy allows the discovery of better separated clusters. Moreover, the algorithm can either be executed until convergence or can be run in iterative steps. In the first case it finds k clusters, while in the second case it runs multiple times with k ranging from 2 to some maximum value k_{max} , and then returns the clustering with largest silhouette coefficient.

Generation of the representative functions. Given the set of clusters C , Procedure GENERATEMAXWEIGHTVECTORS (Algorithm 4, Line 3) generates a representative function ω_j^* for each cluster C_j . Different choices of ω_j^* can lead to different sets of patterns R_j^* , which can contain patterns not relevant for some $\omega_i \in C_j$, as well as missing out patterns relevant for some other $\omega_l \in C_j$. However, as stated in the following theorem, we resort to take the *maximum* among the weights to prevent missing any relevant pattern:

THEOREM 4.1. *Given a cluster C_i , and a MNI-compatible scoring function f , a complete set of relevant patterns for C_i can be mined using the representative function ω_i^* defined as $\forall e \in E, \omega_i^*(e) = \max_{\omega_j \in C_i} \omega_j(e)$.*

PROOF. By definition, only the subgraphs that satisfy the constraints on the weights through the scoring function f can contribute to the score of a pattern. Moreover, the larger the weights of a subgraph, the higher the chances that such subgraph fulfill those constraints. Since the function ω_i^* assigns to each edge $e \in E$ the largest weight among those of the weighting functions in the cluster C_i , i.e., $\forall \omega_j \in C_i, \omega_i^*(e) \geq \omega_j(e)$, the chances that a matching subgraph contributes to the score of a pattern is higher for ω_i^* than for any $\omega_j \in C_i$. It follows that $\forall \omega_j \in C_i, f(P, \omega_i^*) \geq f(P, \omega_j)$, so if a pattern is relevant for some $\omega_j \in C_i$, it is also relevant for ω_i^* . Thus, the set of mined patterns is complete. \square

Given the sets of relevant patterns R_1^*, \dots, R_k^* discovered by Algorithm 1 using the representative functions $\omega_1^*, \dots, \omega_k^*$, we create a pattern set A_i for each function ω_i using the patterns in

the set R_j^* for $j \leq k$. $\omega_i \in C_j$, i.e., each function ω_i receives the set of relevant patterns of the cluster to which it belongs.

4.2 Quality of REsUM approximate

The REsUM *approximate* algorithm reduces the problem of mining patterns in graphs with m weights on each edge to finding k sets of relevant patterns R_j^* , with $k \ll m$. The quality Q of the solution can be measured in different ways, according to the requirements of the user or the application. The most common quality measure used in the literature is the accuracy, which is defined in terms of precision and recall. In our case, since Theorem 4.1 ensures a total recall, we consider the average precision of the sets A_i with respect to the exact sets R_i :

$$Q = \frac{1}{m} \sum_{i=1}^m |R_i \cap A_i| / |R_i| \quad (1)$$

The quality Q can be measured also in terms of the average distance between the patterns in the sets R_i and those in the sets A_i . As shown in Section 7, the distance between two patterns can be calculated using the normalized Levenshtein distance, and the distance between two pattern sets as the average normalized Levenshtein distance among the pairs of closest patterns in the two sets. According to this measure, A_i is a good solution for ω_i if the patterns in A_i have structure and labels similar to the patterns in R_i .

5 PATTERN EVALUATION

A number of scoring function satisfying properties **P1**, **P2**, and **P3** can be proposed and implemented in Procedure ISVALID and RELEVANCESCORE in Algorithm 2 and 3. Nevertheless, to demonstrate the flexibility of our framework, we propose here four different scoring functions that can be used to assess the relevance of a pattern in a weighted graph. They are called *ALL*, *ANY*, *SUM* and *AVG*. We chose these functions because of their intuitive semantics and their suitability for various scenarios that may pose different requirements or provide a different interpretation of the edge weights. Moreover, as they are defined by the MNI support of the pattern over a specific restriction of its support set, they are *MNI-compatible* by definition, and thus they preserve the apriori property.

ALL, *ANY*, *SUM* and *AVG* differ in the choice of which subgraphs they include in the support sets of the patterns P and in how they aggregate the edge weights of such subgraphs. In particular, *ALL*, *ANY*, and *SUM* rely on an additional system-dependent parameter, called relevance threshold α , that is used to select the subgraphs that contribute to the *score*, while *AVG* is parameter-free.

In the following we provide a formal definition of the four scoring functions.

ALL. The *ALL* score considers only the subgraphs whose edge weights are larger than the threshold α as valid appearances of a pattern P . Specifically, the *ALL score* of P is its MNI support computed over the restricted set of appearances $S'_G(P) = \{g \mid g = \langle V_g, E_g, \ell, \omega \rangle \wedge g \in S_G(P) \wedge \forall e \in E_g, \omega(e) > \alpha\}$, i.e., $f_{ALL}(P, G) = \min_{v_P \in V_P} |\mathcal{N}(G, v_P) \upharpoonright_{S'_G(P)}|$, where $\mathcal{N}(G, v_P) \upharpoonright_{S'_G(P)} = \{v \mid v \in V \wedge \exists g \in S'_G(P), \phi_g^P(v) = v_P\}$ is the restriction of $\mathcal{N}(G, v_P)$ to the subset $S'_G(P) \subseteq S_G(P)$.

In graphs like protein-to-protein interaction networks, this *score* retrieves patterns characterized by an overall confidence greater than a certain value.

ANY. The *ANY score* takes into account only the appearances of a pattern having at least one edge with weight above the threshold α . Hence, the *ANY score* of P is the MNI support of P over the set of appearances $S'_G(P) = \{g \mid g = \langle V_g, E_g, \ell, \omega \rangle \wedge g \in S_G(P) \wedge \exists e \in E_g. \omega(e) > \alpha\}$, i.e., $f_{ANY}(P, G) = \min_{v_P \in V_P} |\mathcal{N}(G, v_P) \upharpoonright_{S'_G(P)}|$.

This score is suitable especially for the cases in which only partial weights are available (e.g., product reviews for some product), to find patterns that are overall interesting (e.g., the entire transaction comprising the product), as well as super-patterns around relevant core structures.

By definition, the *ANY score* of P is always equal or larger than its *ALL score*, as any appearance of P considered by f_{ALL} is considered also by f_{ANY} , while in general, the opposite is not true. For example, given the graph in Figure 2 and the relevance threshold $\alpha = 0.4$, the subgraph $g : [1]-A-[2]-C-[4]$ does not contribute to the *ALL score* of $P : [v_1]-A-[v_2]-C-[v_3]$, but contributes to its *ANY score*.

SUM. For the *SUM score* of P , a subgraph g contributes if the sum of its weights is larger than the threshold α . The restricted support set obtained in this way is $S'_G(P) = \{g \mid g = \langle V_g, E_g, \ell, \omega \rangle \wedge g \in S_G(P) \wedge \sum_{e \in E_g} \omega(e) > \alpha\}$. The MNI support over this set is the *SUM score* of P : $f_{SUM}(P, G) = \min_{v_P \in V_P} |\mathcal{N}(G, v_P) \upharpoonright_{S'_G(P)}|$.

This score accounts for the overall pattern weight in scenarios like money transactions, where it is beneficial to sum each single contribution in order to judge the complete value of a structure.

Note that if an appearance of P has some weight greater than α , then the sum of all its weights is at least α , and therefore $f_{SUM}(P, G) \geq f_{ANY}(P, G)$. For example, all the appearances considered by *ANY* in computing the score of $P : [v_1]-A-[v_2]-A-[v_3]$ for $\alpha=0.4$ in Figure 2 are considered also by *SUM*, whereas the subgraph $g : [3]-A-[4]-A-[8]$ contributes to the *SUM score* only.

AVG. In contrast to the previous scoring functions, the *AVG score* is not defined in terms of the minimum cardinality among some node sets of the pattern, but in terms of the relative weights of its appearances. In general, the score of a pattern P can be a function of the sum of the weights of the subgraphs in its support set, and this is called the *weighted support (WSUP)* of P . In particular, WIGM [43] proposes a measure called *normalized weighted support (NWSUP)*, which is the weighted support of P divided by its size $|E_P|$, i.e., $NWSUP(G, P) = WSUP(G, P)/|E_P|$. Nevertheless, this scoring function is not MNI-compatible. In order to guarantee the apriori property and be consistent with the other MNI-compatible scoring functions, we compute $WSUP(G, P)$ by first retaining, for each edge set $\mathcal{E}(G, e_P)$ with $e_P \in E_P$, the set $\mathcal{E}(G, e_P) \upharpoonright_\mu$ of μ edges with largest weight, and then summing up all those weights, i.e., $WSUP(G, P) = \sum_{e_P \in E_P} \sum_{e \in \mathcal{E}(G, e_P) \upharpoonright_\mu} \omega(e)$. Setting μ to be the MNI support of P we guarantee that the *AVG score* is bounded by the MNI support, as stated in the following theorem:

THEOREM 5.1. *Given a graph $G: \langle V, E, \ell, \omega \rangle$, a pattern P , and an edge $e \in E$, it holds that $f_{AVG}(P \diamond e, G) \leq MNI(P, G)$, where $P \diamond e$ is an extension of P with $E_{P \diamond e} = E_P \cup \{e\}$.*

PROOF. Since the MNI support has the apriori property [6], $MNI(P \diamond e, G) \leq MNI(P, G)$. By definition, the pattern $P \diamond e$ has the maximum normalized weight $f_{AVG}^*(P \diamond e, G)$ when all the edges in $\mathcal{E}(G, e) \upharpoonright_\mu$ have weight 1, and hence each subgraph contributes with a total weight of $(|E_P| + 1)$. In this case, $f_{AVG}^*(P \diamond e, G) =$

$MNI(P \diamond e, G) \cdot (|E_P| + 1)/(|E_P| + 1)$, and thus $f_{AVG}(P \diamond e, G) \leq f_{AVG}^*(P \diamond e, G) = MNI(P \diamond e, G) \leq MNI(P, G)$. \square

According to this theorem, although *AVG* does not have the apriori property, the *AVG score* of a pattern is at least bounded by the frequency of its sub-patterns, making it MNI-compatible and allowing early pruning during the pattern search. In fact, if the MNI support of P is lower than τ , then all its super-patterns can be discarded. On the other hand, $f_{AVG}(P \diamond e, G)$ can be higher than $f_{AVG}(P, G)$ even though the frequency of $P \diamond e$ is lower, because the weights of the edges in $\mathcal{E}(G, e) \upharpoonright_\mu$ can be so large that they compensate for the lower frequency. For example, the *AVG score* of $P : [v_1]-C-[v_2]$ in the graph G in Figure 2 is 0.6, because $MNI(P, G) = 1$ and $\mathcal{E}(G, C) \upharpoonright_1 = \{(1, 4)\}$. Instead, the *AVG score* of $P : [v_1]-C-[v_2]-B-[v_3]-A-[v_4]$ is 0.8, because $\mathcal{E}(G, C) \upharpoonright_1 = \{(1, 4)\}$, $\mathcal{E}(G, B) \upharpoonright_1 = \{(1, 3)\}$, and $\mathcal{E}(G, A) \upharpoonright_1 = \{(3, 5)\}$.

Implementation. To implement *ALL*, *ANY*, and *SUM* in our framework, function `isVALID` checks every match g of P in its support set, by comparing its edge weights against the relevance threshold α , according to the corresponding definition of $S'_G(P)$. Then, Procedure `RELEVANCESCORE` computes the MNI support over the support set $S'_G(P)$. On the other hand, for the *AVG score*, Procedure `isVALID` returns always *True*, while Procedure `RELEVANCESCORE` calculates the normalized sum of the top- k edge weights of every pattern edge, where $k = \min_{v \in V_g} |D_v|$.

6 RELATED WORK

We survey the main solutions for pattern mining in *graph databases*, *single graphs*, and *probabilistic graphs*. While previous work has tackled the problem of pattern mining in weighted graphs to a certain extent, no solution has been proposed for pattern mining in multi-weighted graphs.

Graph databases. Graph databases are collections of graphs such as chemical compounds, transactions, and workflows. Two main approaches have been proposed for pattern mining in *unweighted* collections of graphs: apriori-based methods, and pattern-growth methods. The *apriori-based* approaches generate frequent structures incrementally, by merging smaller frequent patterns [24]. Pattern-growth methods, on the other hand, generate one structure at a time, expanding each pattern in a depth-first fashion [17, 41].

In *weighted* graphs, a few pattern-growth methods have been recently introduced [19] to embody weights into the support measure. Additionally, WFSM-MR [3] further extends such approaches in a distributed manner on top of the MapReduce framework.

Nevertheless, frequent pattern mining in graph databases employs a support measure, i.e., the number of graphs containing a specific pattern, that cannot be used to mine patterns in large graphs, as each pattern would have a support equal to 1 or 0.

Single Large Graphs. Pattern mining in large graphs requires the support measure to be adjusted to account for edges shared by multiple subgraphs [6]. To this end, alternative support measures satisfying the apriori property have been proposed, alongside efficient algorithms using such measures. SUBDUE [15] is the first pattern mining algorithm in single graphs and adopts an approximate greedy strategy based on the Minimum Description Length (MDL). Other support measures include the maximum number of edge-disjoint matchings [38], the Maximum Independent Set (MIS) support [25], and the Harmful Overlap (HO) [11] support.

Nonetheless, the latter two measures require NP-complete problems to be solved, rendering them unsuitable in many practical scenarios. In contrast, the Minimum Image-based (MNI) support can be computed efficiently [11]. This measure is used by GraMi [10] and its parallel extension ScaleMine[1], which optimize the computation of the frequent patterns via a constraint satisfaction problem approach. Yet, as opposed to the problem we tackle in this work, GraMi and all the support-based approaches disregard weights on the edges of the graph and do not generalize to the case of multi-weights.

The first work on *weighted* large graphs is WTMaxMiner [12]. However, WTMaxMiner restricts the problem to mining *path* patterns, which can be efficiently discovered as opposed to subgraphs. To the best of our knowledge, WIGM [43] is the only work that deals with weighted pattern mining in large graphs, defining the importance of a pattern as the average weight over its appearances. Although weighted patterns do not naturally possess the apriori property, WIGM adopts a weaker pruning strategy based on the so-called *1-extension property*. Differently from WIGM, our solution, ReSuM is scalable and efficient since it uses measures (a.k.a. scoring functions) that satisfy the apriori property and are based on the MNI support. Additionally, ReSuM is a more general framework that supports multi-weighted graphs, as well as a broad family of scoring functions, showcasing the WIGM support measure as one example (see Section 5).

Uncertain graphs. Uncertain graphs include existence probabilities for edges or nodes of the graph. To some extent, uncertain graphs can be seen as a special case of weighted graphs in which probabilities arises, for instance, from random walk approaches, and represent the likelihood that an edge exists between two nodes. Few works have been proposed to mine frequent patterns in uncertain graphs [7, 18, 26, 32, 40, 44]. As opposed to weighted graphs, support measures for uncertain graphs must consider the uncertainty in the edges and compute the support as an expected value. Moreover, the time complexity of mining in such graphs is exponential in the worst case, since any edge can either exists or not, and hence all the possible combinations must be considered.

7 EXPERIMENTS

We experimentally show how the patterns found with ReSuM differ from those returned by frequency-based methods, thus proving the importance of our approach in pruning irrelevant patterns that are merely frequent. We also compare the scalability of our exact algorithm with the performance of our approximate algorithm. The results demonstrate that ReSuM *approximate* allows faster response time, yet retaining good accuracy in terms of the patterns returned.

Datasets. The experiments were performed on four real datasets of different sizes. Table 1 shows their characteristics, reporting the number of vertices $|V|$, edges $|E|$, and labels $|\mathcal{L}|$; the minimum, average, and maximum node degree; and the minimum, median, average, and maximum edge label frequency. For the AMAZON dataset we report statistics for both edge (top) and node labels (bottom). We also report the default frequency (τ) and relevance (α) values used in the experiments (unless otherwise stated).

- CITESEER [10], is a graph representing Computer Science publications and citations between them. The labels on the edges indicate the area in which the two papers were published (e.g., a database conference).

dataset	$ V $	$ E $	$ \Sigma $	degree		label frequency		τ	α
				min/avg/max	min/med/avg/max				
CITSEER	2.1k	3.6k	21	1/3.5/99	15/55/174.7/988	95	.05		
FREEBASE-T	7.2k	10k	40	1/2.8/504	3/70/251.3/2886	90	.05		
FREEBASE-C	16.7k	26k	77	1/3.2/1082	1/66/348.5/4861	155	.05		
AMAZON	163k	296k	4 1710	1/3.6/1072	2k/12k/30k/113k 1/1/95/142k	130	.05		

Table 1: Datasets and default τ , α parameters.

- FREEBASE-T and FREEBASE-C are directed subgraphs extracted from the knowledge graph FreeBase¹, which is a database collecting structured information about real-world entities like people, places and things for various topics. We obtained the two samples by restricting the graph to the topic *travel* and *computer* respectively, and then taking the largest weakly connected component in the restriction.

- AMAZON² [14] is a directed graph representing items, purchases, and user ratings. We considered the subgraph of electronic products, in which every node represents a product, a category, or a brand, and a link represents items bought together, bought in subsequent transactions, or viewed on the website one after the other. Weights represent individual user review scores (from 1 to 5), and we considered only users with more than 100 reviews. Given the sparsity of the weights, we used Personalized PageRank to spread the user preferences to products other than those they rated, as it is a standard technique for recommendations [2]. In this way we obtained weights not only for the items reviewed, but also for the most related items. Each edge weight is actually computed as the average between the PageRank value of its endpoint nodes.

Experimental setup. ReSuM is implemented in Java 1.8 on top of the constraint satisfaction problem presented in GRAMI [10] whose code was kindly provided by the authors³. The code of our implementation and all the datasets we used are publicly available⁴. We also compare with a frequent pattern mining approach (FREQ) based on GRAMI, which is also implemented in Java 1.8. All experiments were run on a 24 Cores (2.40GHz) Intel Xeon E5 – 2440 with 188Gb RAM with Linux 3.13.

Generating the weights. Since we had real weights only for the AMAZON graph, to test the scalability of our method with a larger number of weighting functions, for the other datasets we created synthetic weights based on the results of a user study we conducted on the Crowdfunder⁵ platform. We extracted a sample from the FreeBase knowledge base, restricting the domain of the edge labels to five topics (Music, Books, Celebrities, Movies, and Sport). Then we asked the users to rate each graph edge (i.e., fact) according to their preferences, using a relevance value between 1 and 5. Once collected the relevance values from 123 users, we modeled the distribution of the edge weights with respect to the number of facts. We found that the edge weights, after normalization, are distributed as a Gaussian with mean 0.452 and variance 0.02. In addition, we noted that, on average, a user rated above 0 between 10% and 20% of the labels, and thus we concluded that real graph weights are usually quite sparse. Therefore, we uniformly subset edge labels according to our findings and generated weights normally distributed in $[0, 1]$.

Furthermore, in order to evaluate the performance of ReSuM and ReSuM *approximate* with different weight distributions, we

¹developers.google.com/Freebase/data

²jmcasley.ucsd.edu/data/amazon/

³github.com/ehab-abdelhamid/GraMi

⁴https://github.com/lady-bluecopper/ReSuM

⁵www.crowdfunder.com

top-k	FREEBASE-C				FREEBASE-T			
	ALL	ANY	SUM	AVG	ALL	ANY	SUM	AVG
1	0.6	0.6	0.6	0.86	0.5	0.5	0.5	1
3	0.43	0.43	0.43	1	0.45	0.33	0.33	1
10	0.44	0.49	0.49	1	0.8	0.66	0.66	1

Table 2: Quality of FREQ vs RESUM on the top-k patterns.

generated sets of synthetic edge weights, varying a *focus* parameter representing the ratio of weighted edges for each edge label. The edge weights were sampled from a normal distribution $\mathcal{N}(\mu, \sigma^2)$ and a *Beta*(α, β) distribution, hence allowing us to prove the effectiveness of our algorithms under normally distributed weights and exponentially distributed weights. We set $\mu = 0.5$ and $\sigma = 0.25$ for the normal distribution and $\alpha = 0.7, \beta = 5$ and $\beta = 0.7, \alpha = 5$, for the *Beta* distribution. The two choices of the parameters for the *Beta* distribution represent two extreme of an exponential behavior: the former concentrates the probability mass on low weights, the latter on large weights. The focus parameter takes values in $\{0.5, 0.8\}$ for the normal distribution and in $\{0.25, 0.5, 0.75, 1\}$ for the *Beta* distribution.

7.1 Frequent vs Weighted Pattern Mining

We compared the patterns returned by a frequent pattern mining algorithm (FREQ) and our algorithm RESUM to validate our claim that frequent pattern mining returns a large number of low-weight patterns, which, instead, are correctly discarded in relevant pattern mining. Unless otherwise stated, we report the average of 10 different randomly sampled weighting functions. In particular, these weights were sampled from a normal distribution using focus 0.5, as previously described.

Figure 5 reports the average number of patterns found using different scoring functions on the four datasets, with default parameters, as shown in Table 1. We observe that FREQ returns patterns, at least half of which are irrelevant with respect to any of the four scoring functions. As expected, in all the datasets, ANY and SUM return more patterns than ALL and AVG, due to the less restrictive conditions on the weights. On the other hand, AVG returns a low number of patterns, mainly because more than 50% of the edges have low or zero weight. Therefore, AVG is particularly suited in biological or chemical datasets, where weights are uniformly distributed in the entire graph.

We now discuss quality (Table 2), number of patterns, and running time of RESUM compared to FREQ, when varying relevance (α) and frequency (τ) threshold (Figure 3 and 4). Due to space limits, we report results for two datasets (FREEBASE-C and FREEBASE-T); however, we observe similar results also on the other datasets. In particular, as an example, within the top-5 frequent patterns in the AMAZON graph, we found that the most frequently bought products are Sony appliances, but some relevant patterns actually involve Nikon products. This result shows that Sony products are popular but not interesting for all the users.

Quality of FREQ vs RESUM. Table 2 shows the quality of the patterns discovered by FREQ, measured on the k most frequent patterns. We selected 10 random weighting functions and mined the relevant patterns for each of them. The quality of FREQ is measured as the average Jaccard similarity between the top- k frequent patterns and the top- k relevant patterns. As expected, frequency is a bad predictor of relevance, since most of the relevant patterns are not in top- k frequent patterns. Notably, for AVG the quality is higher mostly due to the small or null number of patterns returned, as reported in Figure 5.

Relevance threshold (α). Recall that the relevance threshold α is a system-dependent parameter set only for ALL, ANY, and SUM. It can be easily tuned on demand and strongly affects the number of patterns (Figure 3(a) and Figure 3(b)), because the larger the value of α , the smaller is the number of appearances that are considered valid, and thus the smaller is the total number of relevant patterns mined. We observe that with $\alpha > 0$ the number of relevant patterns is less than half of the number of the frequent ones. This behavior reflects the characteristics of the weights in the datasets, as half of the edges have zero weight. Moreover, for FREEBASE-T SUM, being the most lenient scoring function, returns patterns even in the restrictive cases when $\alpha > 0.5$ (Figure 3(b)). Finally, since AVG does not depend on α , it always returns the same patterns.

Figure 3(c) and Figure 3(d) show that the threshold α affects the running time of RESUM mostly when ALL is used, as this function can prune the irrelevant patterns earlier in the process. In fact, an occurrence of a pattern is discarded and not included in the support set of any extension of the pattern, as soon as one edge weight is found to be below α . On the other hand, for all the other scoring functions, the extension of an invalid occurrence of a pattern can be valid for some super-pattern, and therefore cannot be discarded until all its edge weights have been examined. As a consequence, the running time of the algorithm is almost unaffected by α .

Frequency threshold (τ). Figure 4 reports the behavior of RESUM and FREQ when varying the frequency threshold τ . We performed preliminary tests to decide a reasonable range of values $[\tau_{min}, \tau_{max}]$ for each dataset. In particular, the τ_{min} corresponds to the smallest value that allowed FREQ to terminate the computation within 48 hours, and τ_{max} is the maximum value returning a non-empty set of frequent patterns. The choice of different ranges for each dataset is consistent with previous researches [10] and reflects the observation that pattern frequency is dataset-dependent, while relevance is user-dependent.

As we can see in Figure 4(a) and Figure 4(b), the number of frequent patterns decreases almost linearly with τ , and consequently the number of relevant patterns decreases as well. Regarding the performance, as opposed to the relevance threshold, the frequency threshold always alters the computation time, since higher values lead to an early pruning of many patterns, and thus the algorithm terminates earlier. Moreover, Figure 4(c) and Figure 4(d) show that when τ takes low values (i.e. between 150 and 180), RESUM runs up to two orders of magnitude faster in both the datasets. Finally, as previously noted, ALL performs significantly better than the other scoring functions.

7.2 Multiple Weighting Functions

We tested the scalability of RESUM in the case of multiple weighting functions, varying their number between 50 and 50,000. Similarly, we also measured time and quality of RESUM approximate. Nevertheless, in the following we do not further discuss and report the number of patterns retrieved for each weighting function and each scoring function, since these results are consistent with what reported in the single edge weight case.

Time. Figure 6 shows how the number of weighting functions affects the running time. Here we report the performance obtained when the weights were generated following a normal distribution with focus 0.5. In Figure 6(a) we present the comparison between RESUM and the brute-force (BF) approach, which computes the patterns for each weighting function separately. While BF scales

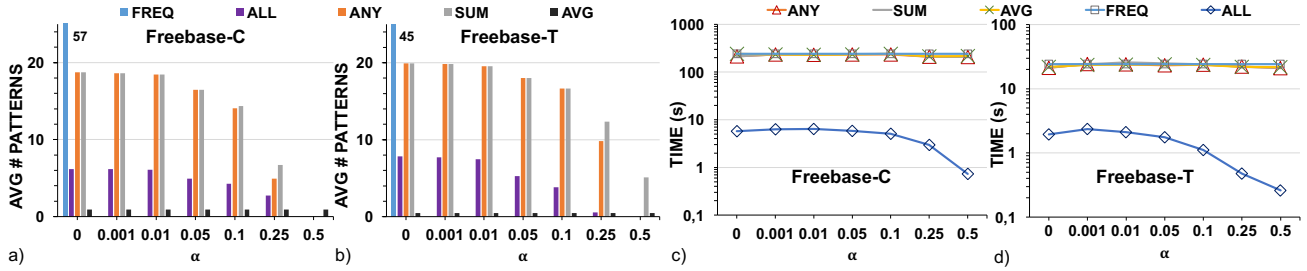


Figure 3: Varying α : number of patterns (left) and running time (right) in FREEBASE-C (a,c) and FREEBASE-T (b,d).

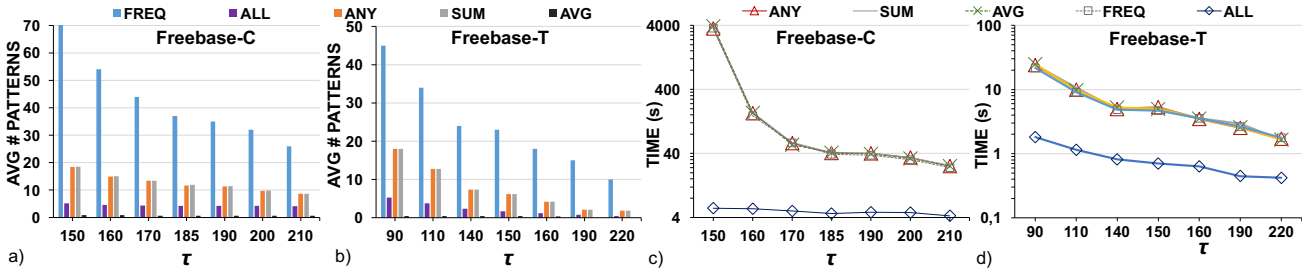


Figure 4: Varying τ : number of patterns (left) and running time (right) in FREEBASE-C (a,c) and FREEBASE-T (b,d).

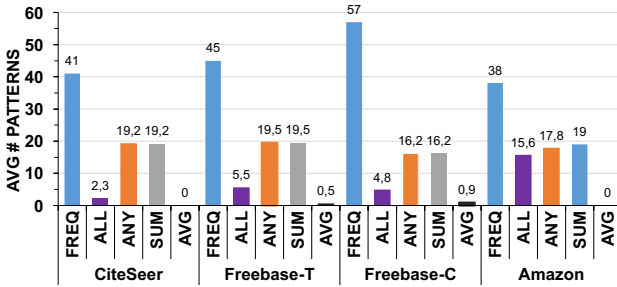


Figure 5: Number of patterns found in each dataset, using different scores and default parameters.

W	average pattern edit distance			
	ALL	ANY	SUM	AVG
50	0.195	0.069	0.069	0.627
500	0.192	0.062	0.062	0.618
5000	0.203	0.053	0.053	0.609
50000	0.204	0.052	0.051	—

Table 3: Quality of ReSUM *approximate* in FREEBASE-T.

clustering	average pattern edit distance							
	FREEBASE-C				FREEBASE-T			
	ALL	ANY	SUM	AVG	ALL	ANY	SUM	AVG
A-POST	0.28	0.07	0.07	0.45	0.2	0.07	0.07	0.7
BUCK	0.27	0.07	0.07	0.39	0.2	0.06	0.06	0.62

Table 4: Quality of ReSUM *approximate* using BUCK and A-POST clustering in FREEBASE-T and FREEBASE-C.

linearly with the number of weighting functions, the running time of ReSUM is nearly constant with 5000 functions, and slowly increases as the number of edge weights approaches 50000. As a drawback, the memory requirement grows linearly with the number of weights for both algorithms.

In Figure 6(b) and Figure 6(c) instead, we compare ReSUM and ReSUM *approximate*. For these set of experiments, we generated the representative functions by first clustering the weighting functions using the bucket-based strategy. The clustering phase is performed as a preprocessing and not reported, since it is

agnostic to the choice of the various thresholds and depends solely on the clustering algorithm (e.g., k -means, hierarchical, or spectral). In particular, we tried numbers of buckets b of different orders of magnitude and proportional to the frequency of the edge labels in the graph. Then, we run k -means using different k to study the impact of the number of clusters on the quality and the running time of ReSUM *approximate*. Finally, we set the default value of b of each dataset to the number of buckets that allowed the algorithm to use at least one order of magnitude less memory than those consumed using the full-vector strategy, i.e., 12 buckets for FREEBASE-T, 16 for FREEBASE-C, and 10 for CITESEER.

We observe that ReSUM becomes impractical as the number of weighting functions increases. As a matter of fact, when AVG is used, ReSUM exhausts the available memory, hence returning no patterns. This behavior reflects the characteristics of AVG, which requires the algorithm to exhaustively search for all the occurrences of a pattern before computing its score. In contrast, ReSUM *approximate* terminates the computation. On the other hand, when ANY is used, ReSUM is able to return the relevant patterns; however, ReSUM *approximate* outperforms the exact algorithm again, taking nearly constant time to terminate. In conclusion, in all the cases of large numbers of weighting functions, ReSUM *approximate* performs better than ReSUM by at least one order of magnitude.

Quality of ReSUM *approximate*. As mentioned in Section 4, we measure the quality of ReSUM *approximate* in terms of the average distance between the patterns it returns (sets A_i) and those returned by ReSUM (sets R_i). We define the distance between two patterns as the minimum number of edges that should be added or removed from the first to transform it into the second. Thus, the average distance between the two sets of patterns $\{A_1, \dots, A_m\}$ and $\{R_1, \dots, R_m\}$ measures the average number of operations required to transform a pattern in A_i to a pattern in R_i . We recall that our method is complete, and therefore no relevant pattern is missing. However, ReSUM *approximate* may return spurious patterns, which are patterns not relevant for any function in the cluster. Computing the distance between the

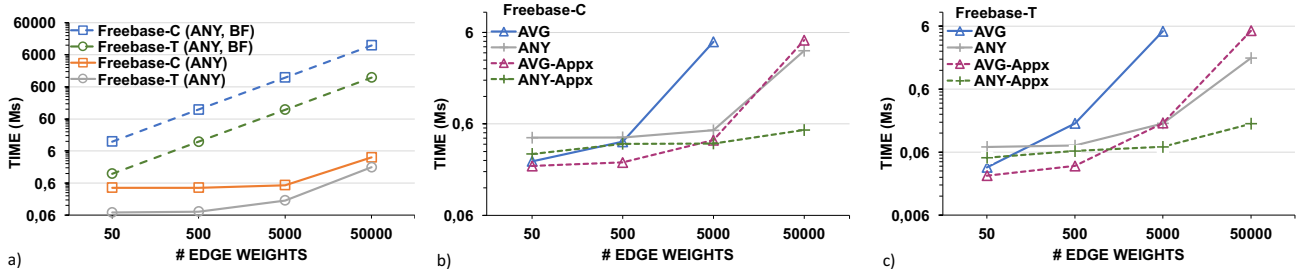


Figure 6: Varying number of edge weights in FREEBASE-C and FREEBASE-T: running time of ReSUM and brute-force approach (BF) with ANY (a); and running time of ReSUM and ReSUM *approximate* with ANY and AVG (b, c).

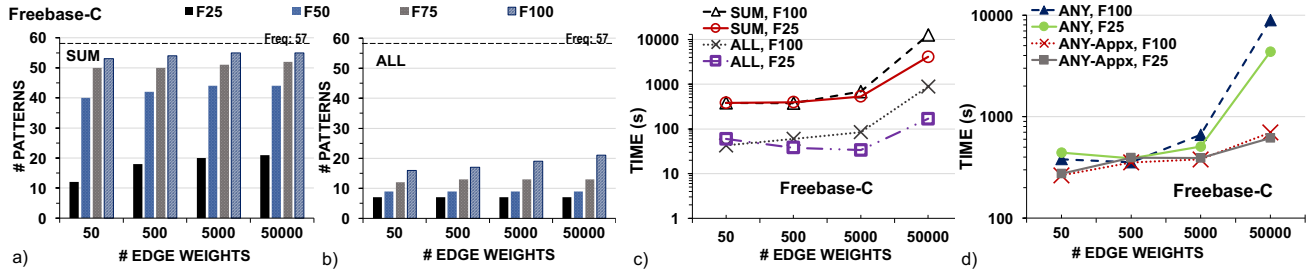


Figure 7: Varying *focus* in FREEBASE-C: number of patterns using SUM (a) and ALL (b) with focus between 0.25 (F25) and 1 (F100); and running time of ReSUM and ReSUM *approximate* with $Beta(0.7, 5)$ weights with focus 0.25 (F25) and 1 (F100), using SUM, ALL (c), and ANY (d).

		average precision																			
		$Beta(0.7, 5)$								$Beta(5, 0.7)$								$N(0.5, 0.25)$			
		ALL				SUM				ALL				SUM				ALL		SUM	
$ W $		0.25	0.5	0.75	1	0.25	0.5	0.75	1	0.25	0.5	0.75	1	0.25	0.5	0.75	1	0.5	0.8	0.5	0.8
50		0.22	0.20	0.21	0.26	0.17	0.53	0.74	0.91	0.19	0.23	0.42	1	0.34	0.73	0.94	1	0.15	0.18	0.36	0.54
500		0.21	0.21	0.24	0.27	0.18	0.53	0.74	0.91	0.21	0.24	0.42	1	0.36	0.73	0.94	1	0.18	0.20	0.44	0.57
5000		0.21	0.22	0.24	0.28	0.19	0.54	0.75	0.91	0.21	0.25	0.43	1	0.36	0.74	0.95	1	0.22	0.20	0.49	0.59
50000		0.21	0.22	0.25	0.28	0.19	0.54	0.75	0.91	0.21	0.25	0.44	0.99	0.36	0.74	0.95	1	0.22	0.21	0.51	0.59

Table 5: Quality of ReSUM *approximate* with ALL and SUM on FREEBASE-C, with $Beta(\alpha, \beta)$ and normal $N(\mu, \sigma^2)$ weights generated using *focus* values in $\{0.25, 0.5, 0.75, 1\}$ and $\{0.5, 0.8\}$ respectively.

two pattern sets allows us to understand how much a spurious pattern, on average, differs from the patterns that are actually relevant for some weighting function in the cluster. Table 3 reports the distances obtained using the four scoring functions in FREEBASE-T. Here, ANY and SUM exhibit the best quality; ALL performs reasonably good, despite being more restrictive and therefore more sensitive to the approximation based on the maximum edge weights. On the other hand, when AVG is used, the quality of the answer is quite poor. Nevertheless, this behavior is due to the extremely low number of patterns this scoring function considers interesting, which skews the computation of the pattern set distance. Note that, we do not report any value for the case of 50000 weighting functions with AVG, since the algorithm exhausted all the available memory and did not terminate. We conclude that, the additional patterns returned by ReSUM *approximate* are indeed closely related to the relevant patterns of each individual weighting function.

Finally, we tested the capability of our bucket-based clustering (BUCK in short) to correctly identify groups of similar weighting functions. To this end, we compared the quality of the results mined using BUCK in the creation of the feature vectors of the weighting functions, with the quality measured using a *ground-truth* clustering (A-POST in short). The A-POST clustering was created using the sets of relevant patterns R_1, \dots, R_m as feature vectors of $\omega_1, \dots, \omega_m$, and then running a k -medoid algorithm.

We regard it as a ground-truth clustering, because it is obtained knowing what makes two weighting functions really similar, i.e. their relevant patterns, and maximizing the intra-cluster similarity. Table 4 reports the comparison between A-POST and BUCK on FREEBASE-C and FREEBASE-T. We recall that lower values mean higher quality, as they indicate distances. We can see that we experience a quality comparable with that obtained using A-POST, and thus we can conclude that our clustering technique is indeed effective.

Impact of the weights. For the experiments presented above, we weighted the Amazon graph using real weights, and the FREEBASE-T, FREEBASE-C, and CITESEER graphs with synthetic weights generated according to the results of our user study. The common feature of these two kinds of weights is that they are highly sparse. It is worth studying whether weights following other distributions or that are denser, affect the performance of our algorithms. To this end, we performed an additional set of experiments using weighting functions generated following a $Beta(5, 0.7)$, a $Beta(0.7, 5)$ and a normal distribution with different densities (*focus*), as described at the beginning of Section 7.

One would expect that, with higher densities, the cost of the computation would be higher too. Although these expectations are reasonable, in the following we show that the behavior of ReSUM and ReSUM *approximate* is consistent with what observed

in the case of sparse weights. Figure 7(a) and Figure 7(b) report the average number of patterns found using *SUM* and *ALL*, with weights generated using a *Beta*(0.7, 5) distribution with focus varying between 0.25 and 1 (i.e., all edges have weight > 0). Comparing these results with those in Figure 5 when *SUM* is used, we can see that the number of relevant patterns is largely affected by the presence of more (or all) edges with non-null weight, meaning that the patterns mined are actually many more. On the other hand, when *ALL* is used, *ReSUM* still finds a larger number of relevant patterns, but the increment is not as large as in the *SUM* case.

Regarding the running time, Figure 7(c) and Figure 7(d) show that the two algorithms behave accordingly to what already seen in the previous experiments, meaning that the fact there more patterns are mined do not downgrade the performance heavily.

Finally, Table 5 displays the quality of *ReSUM* in terms of average precision, as defined in Equation 1. As we can see, our approximate algorithm achieves similar quality values no matter which weight distribution is chosen. In addition, the denser the weights in the graph, the higher is the average precision of the pattern sets mined. Intuitively, this is due to the fact knowing a larger number of positive weights allows the clustering algorithm to better detect which weighting functions are similar.

8 CONCLUSIONS

In this paper we considered the problem of mining relevant patterns in weighted graphs. As opposed to the previous graph pattern mining approaches, which are solely based on the frequency of the patterns, our solution assesses the importance of a pattern also in terms of the weights on the edges of its appearances. Then, we proposed four different scoring functions that balance between frequency and weights, while retaining the apriori property, which is a powerful mean to an effective and early pruning of the search space. As a natural extension, we considered the complementary problem of mining patterns in graphs with multiple weights associated to the edges. We devised exact and approximate solutions and proved the effectiveness and efficiency of the algorithms on real datasets. As a future work, we plan to study the theoretical bounds on the clustering quality, and automatic approaches for parameter selection.

REFERENCES

- [1] Ehab Abdelhamid, Ibrahim Abdelaziz, Panos Kalnis, Zuhair Khayyat, and Fuad Jamour. 2016. ScaleMine: Scalable parallel frequent subgraph mining in a single large graph. In *International Conference for High Performance Computing, Networking, Storage and Analysis*. 716–727.
- [2] Charu C Aggarwal. 2016. *Recommender Systems*. Springer.
- [3] Nisha Babu and Anamma John. 2016. A distributed approach to weighted frequent Subgraph mining. In *International Conference on Emerging Technological Trends*. 1–7.
- [4] Dorna Bandari, Shuo Xiang, and Jure Leskovec. 2017. Categorizing User Sessions at Pinterest. *arXiv preprint arXiv:1703.09662* (2017).
- [5] Petko Bogdanov, Misael Mongiovi, and Ambuj K Singh. 2011. Mining heavy subgraphs in time-evolving networks. In *Data Mining (ICDM), 2011 IEEE 11th International Conference on*. IEEE, 81–90.
- [6] Bjorn Bringmann and Siegfried Nijssen. 2008. What is frequent in a single graph?. In *PAKDD*. 858–863.
- [7] Yifan Chen, Xiang Zhao, Xuemin Lin, and Yang Wang. 2015. Towards frequent subgraph mining on single large uncertain graphs. In *2015 IEEE International Conference on Data Mining*. 41–50.
- [8] James C Costello, Mehmet M Dalkilic, Scott M Beason, Jeff R Gehlhausen, Rupali Patwardhan, Sumit Middha, Brian D Eads, and Justen R Andrews. 2009. Gene networks in *Drosophila melanogaster*: integrating experimental data to predict gene function. *Genome biology* 10, 9 (2009), R97.
- [9] L. De Raedt and A. Zimmermann. 2007. Constraint-Based Pattern Set Mining. In *SDM*. 237–248.
- [10] M. Elseidy, E. Abdelhamid, S. Skiadopoulos, and P. Kalnis. 2014. Grami: Frequent subgraph and pattern mining in a single large graph. *PVLDB* 7, 7 (2014), 517–528.
- [11] M. Fiedler and C. Borgelt. 2007. Subgraph support in a single large graph. In *ICDM Workshops*. 399–404.
- [12] R. Geng, X. Dong, P. Zhang, and W. Xu. 2008. WtMaxMiner: Efficient Mining of Maximal Frequent Patterns Based on Weighted Directed Graph Traversals. In *CCIS*. 1081–1086.
- [13] Michael Greenwald and Sanjeev Khanna. 2001. Space-efficient online computation of quantile summaries. In *ACM SIGMOD Record*, Vol. 30. 58–66.
- [14] Ruining He and Julian McAuley. 2016. Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering. In *WWW*. 507–517.
- [15] Lawrence B Holder, Diane J Cook, Surnjani Djoko, and others. 1994. Substructure Discovery in the SUBDUE System.. In *KDD Workshop*. 169–180.
- [16] J. Huan, D. Bandyopadhyay, W. Wang, J. Snoeyink, J. Prins, and A. Tropsha. 2005. Comparing Graph Representations of Protein Structure for Mining Family-specific Residue-based Packing Motifs. *J. Comput Biol.* 12, 6 (2005), 657–671.
- [17] J. Huan, W. Wang, J. Prins, and J. Yang. 2004. Spin: mining maximal frequent subgraphs from graph databases. In *SIGKDD*. 581–586.
- [18] Shawana Jamil, Azam Khan, Zahid Halim, and A Rauf Baig. 2011. Weighted muse for frequent sub-graph pattern finding in uncertain dblp data. In *2011 International Conference on Internet Technology and Applications*. 1–6.
- [19] C. Jiang, F. Coenen, and M. Zito. 2010. Frequent sub-graph mining on edge weighted graphs. In *DAWAK*. 77–88.
- [20] H. Jiang, H. Wang, P. S. Yu, and S. Zhou. 2007. GString: A Novel Approach for Efficient Search in Graph Databases. In *ICDE*. 566–575.
- [21] W. Jiang, J. Vaidya, Z. Balaporia, C. Clifton, and B. Banich. 2005. Knowledge discovery from transportation network data. In *ICDE*. 1061–1072.
- [22] Xin Jin, Chi Wang, Jiebo Luo, Xiao Yu, and Jiawei Han. 2011. LikeMiner: a system for mining the power of ‘like’ in social media networks. In *KDD*. 753–756.
- [23] Minoru Kanehisa and Susumu Goto. 2000. KEGG: kyoto encyclopedia of genes and genomes. *Nucleic acids research* 28, 1 (2000), 27–30.
- [24] M. Kuramochi and G. Karypis. 2001. Frequent subgraph discovery. In *ICDM*. 313–320.
- [25] M. Kuramochi and G. Karypis. 2005. Finding frequent patterns in a large sparse graph. *DMKD* 11, 3 (2005), 243–271.
- [26] Jianzhong Li, Zhaonian Zou, and Hong Gao. 2012. Mining frequent subgraphs over uncertain graph databases under probabilistic semantics. *VldbJ* 21, 6 (2012), 753–777.
- [27] S. P. Lloyd. 1982. Least squares quantization in PCM. *IEEE Trans. Inf. Theory* 28, 2 (1982), 129–137.
- [28] A. K. Mackworth. 1977. Consistency in networks of relations. *Artificial Intelligence* 8, 1 (1977), 99–118.
- [29] Davide Mottin, Matteo Lissandrini, Yannis Velegrakis, and Themis Palpanas. 2016. Exemplar queries: a new way of searching. *Vldb J.* (2016), 1–25.
- [30] Mark EJ Newman. 2004. Analysis of weighted networks. *Physical review E* 70, 5 (2004).
- [31] C. C. Noble and D. J. Cook. 2003. Graph-based Anomaly Detection. In *SIGKDD*. 631–636.
- [32] Odysseas Papapetrou, Ekaterini Ioannou, and Dimitrios Skoutas. 2011. Efficient discovery of frequent subgraph patterns in uncertain graph databases. In *Proceedings of the 14th International Conference on Extending Database Technology*. 355–366.
- [33] J. Pei, J. Han, B. Mortazavi-Asl, and H. Zhu. 2000. Mining Access Patterns Efficiently from Web Logs. In *PAKDD*. 396–407.
- [34] Michael J Shaw, Chandrasekar Subramaniam, Gek Woo Tan, and Michael E Welge. 2001. Knowledge management and data mining for marketing. *Decision support systems* 31 (2001), 127–137.
- [35] Arlei Silva, Wagner Meira Jr, and Mohammed J Zaki. 2012. Mining attribute-structure correlated patterns in large attributed graphs. *PVLDB* 5, 5 (2012), 466–477.
- [36] Q. Song, Y. Wu, and X. L. Dong. 2016. Mining Summaries for Knowledge Graph Search. In *ICDM*. 1215–1220.
- [37] Michael Steinbach, Levent Ertöz, and Vipin Kumar. 2004. The challenges of clustering high dimensional data. In *New Directions in Statistical Physics*. 273–309.
- [38] N. Vanetich, S. E. Shimony, and E. Gudes. 2006. Support measures for graph data. *Data Min. Knowl. Discov.* 13, 2 (2006), 243–260.
- [39] Haixun Wang and Charu C Aggarwal. 2010. A survey of algorithms for keyword search on graph data. In *Managing and Mining Graph Data*. 249–273.
- [40] Di Wu, Jiadong Ren, and Long Sheng. 2017. Uncertain maximal frequent subgraph mining algorithm based on adjacency matrix and weight. *International Journal of Machine Learning and Cybernetics* (2017), 1–11.
- [41] X. Yan and J. Han. 2002. gspan: Graph-based substructure pattern mining. In *ICDM*. 721–724.
- [42] X. Yan, P. S. Yu, and J. Han. 2004. Graph Indexing: A Frequent Structure-based Approach. In *SIGMOD*. 335–346.
- [43] J. Yang, W. Su, S. Li, and M. M. Dalkilic. 2012. WIGM: Discovery of Subgraph Patterns in a Large Weighted Graph. In *SDM*. 1083–1094.
- [44] Zhaonian Zou, Jianzhong Li, Hong Gao, and Shuo Zhang. 2010. Mining frequent subgraph patterns from uncertain graph data. *IEEE Transactions on Knowledge and Data Engineering* 22, 9 (2010), 1203–1218.

Scalable Evaluation of k -NN Queries on Large Uncertain Graphs

Xiaodong Li¹, Reynold Cheng¹, Yixiang Fang¹, Jiafeng Hu¹, Silviu Maniu²

¹The University of Hong Kong, China, ²Université Paris-Sud, France

{xdli,ckcheng,yxfang,jhu}@cs.hku.hk,silviu.maniu@lri.fr

ABSTRACT

Large graphs are prevalent in social networks, traffic networks, and biology. These graphs are often inexact. For example, in a friendship network, an edge between two nodes u and v indicates that users u and v have a close relationship. This edge may only exist with a probability. To model such information, the *uncertain graph model* has been proposed, in which each edge e is augmented with a probability that indicates the chance e exists. Given a node q in an uncertain graph \mathcal{G} , we study the k -NN query of q , which looks for k nodes in \mathcal{G} whose distances from q are the shortest. The k -NN query can be used in friend-search, data mining, and pattern-recognition. Despite the importance of this query, it has not been well studied. In this paper, we develop a tree-based structure called the *U-tree*. Given a k -NN query, the *U-tree* produces a compact representation of \mathcal{G} , based on which the query can be executed efficiently. Our results on real and synthetic datasets show that our algorithm can scale to large graphs, and is 75% faster than state-of-the-art solutions.

1 INTRODUCTION

Graphs are prevalent in social networks [10, 12], traffic networks [11], biological networks [32], and mobile ad-hoc networks [13]. Due to noisy measurements [1], hardware limitation [2], inference models [9], and privacy-preserving perturbation [4, 23], these graphs are inherently uncertain. To model this error, *uncertain graphs* have been studied [1, 6, 15, 19]. In these graphs, each edge is associated with a probability distribution. Figure 1(a) shows the protein-protein interaction (PPI) network as an uncertain graph, where each node denotes a protein, and each edge is an interaction between a pair of proteins. The value on each edge denotes the probability that the interaction exists (called *existential probability*). For instance, the existential probability between nodes A and B is 0.7.

k -NN query. In this paper, we study the efficient evaluation of *k -nearest neighbor* (k -NN) queries on uncertain graphs [1]. Given a node q , the k -NN query returns k nodes with the shortest “distances” from q , based on a distance function (Table 1). The k -NN query can help biologists to perform tasks such as protein complex detection [2, 21], link discovery [17, 32], and protein function prediction [24]. Security experts can also use k -NN queries to design privacy-preserving algorithms to protect nodes in a graph from being identified [4].

Although k -NN queries are useful, the issues of evaluating them efficiently have only been briefly touched, e.g., [19]. Our experiments found that they are also not very efficient on large uncertain graphs. The major reason is that for correct query execution, queries running on the uncertain graph should follow the *Possible World Semantics* (PWS in short). Figure 1(b) shows

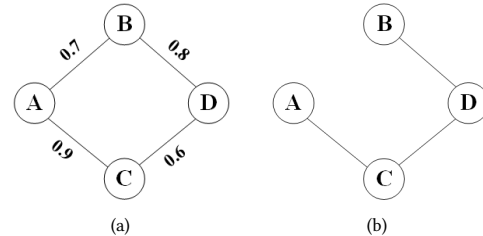


Figure 1: (a) an uncertain graph \mathcal{G} and (b) a possible world from \mathcal{G} with existence probability $0.3 \times 0.9 \times 0.6 \times 0.8 = 0.1296$.

Table 1: Distance functions for uncertain graphs.

Function	Formula
Most-probable [32]	$d_{mp}(s, t) = \arg \max \prod_{e \in PATH(s, t)} p(e)$
Reliability [35]	$d_{re}(s, t) = \sum_{d d < \infty} P_{s, t}(d)$
Median [30]	$d_{me}(s, t) = \arg \max \{ \sum_{d=0}^D P_{s, t}(d) \leq \frac{1}{2} \}$
Majority [30]	$d_{ma}(s, t) = \arg \max_d \{ P_{s, t}(d) \}$
Expected [32]	$d_{ex}(s, t) = \sum_{d=0}^{\infty} P_{s, t}(d) d$
Expected-reliable [35]	$d_{er}(s, t) = \sum_{d d < \infty} d \frac{P_{s, t}(d)}{1 - P_{s, t}(\infty)}$

a *possible world* instance sampled from \mathcal{G} in Figure 1(a). A simple way to evaluate the query is to get the answer from each possible world and then collect all these answers to form the final answer. For example, given a k -NN query of computing the k nearest neighbors from node A in Figure 1(a), we obtain 2^4 possible worlds, then for each of them, we compute the k nearest neighbors, and finally obtain the k nodes which are the closest to A considering all the possible worlds (according to a function in Table 1). Because an uncertain graph has an exponential number of *possible worlds*, a naïve solution can be extremely inefficient. Although existing solutions try to avoid enumerating all the possible worlds, they are still expensive.

The U-tree framework. In this paper, we develop a new indexing framework for k -NN requests, which (i) allows efficient and scalable k -NN query evaluation under the PWS; and (ii) can be easily adapted to different distance functions (e.g., Table 1). As shown in Figure 2, our framework consists of two stages:

- *offline index construction* (Figure 2(a)). Given an uncertain graph \mathcal{G} and a distance function d , we first employ *decomposition techniques* (Step A), which converts \mathcal{G} into a succinct index structure, whose edges are encoded with the probability information of \mathcal{G} [5, 14, 16, 22, 31, 33]. These techniques were not designed for k -NN queries. Hence, we perform *customization* of the index (Step B), which adjusts the information and structure of the index, in order to enhance the performance of k -NN queries.

- *online query evaluation* (Figure 2(b)). This stage is used to evaluate a k -NN request for a node q online. Particularly, we design an efficient query algorithm (Step C) that uses the U-tree developed

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

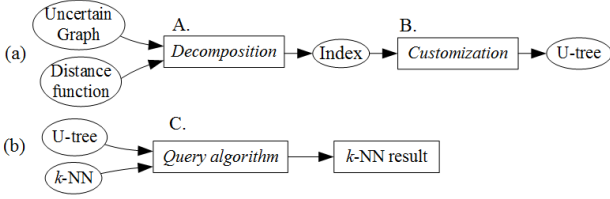


Figure 2: Answering k -NN query with the U-tree.

in the offline stage, which then yields the k nodes that are the closest to q according to distance function d .

Cost model. Upon receiving a k -NN request, the U-tree will generate another uncertain graph g (Step C above), which is a proper subgraph of \mathcal{G} , but contains the probability information essential to answering the k -NN query. Because g is often much smaller than \mathcal{G} , the query execution time can be significantly reduced. To achieve this goal, more information needs to be incorporated to the index during customization, which also renders a larger U-tree (Step B). As we will point out later, there is a trade-off between the U-tree’s size and the query cost. We will discuss a cost model that allows us to balance between these two factors, in order to improve query efficiency without significantly increasing the U-tree size.

Experiments. To evaluate the performance of the proposed methods, we conduct experiments on both real and synthetic datasets. We have also tested different decomposition methods and distance functions in the U-tree framework. The results show that U-tree is superior to the state-of-the-art algorithms, and the query time is significantly reduced by 75%; the overhead of U-tree is only 23% more than the index generated in Step A. Our solution is also scalable to large uncertain graphs with over one million nodes.

Organization. The rest of paper is organized as follows. We review the related works in Section 2. In Section 3, we present the formal definition of the problems, and discuss some basic techniques. We present the U-tree framework in Section 4. In Section 5, we present our experiment results. Section 6 concludes.

2 RELATED WORK

We review the existing queries for uncertain graphs, and then discuss the probabilistic distance functions.

2.1 Queries for Uncertain Graphs

In recent years, the k -NN query for uncertain graphs has received plenty of attention [19, 29, 30]. As mentioned before, to answer queries on uncertain graphs, the Possible World Semantics (PWS) are often used, which assumes that an uncertain graph can be expressed as a number of graph instances. Because there is an exponential no. of possible world instances, three methods are proposed in the literature: (1) The first one finds some representative deterministic graphs from the uncertain graph and answer the queries based on them [28]. (2) The second one is to reduce the size of the uncertain graph by removing some weak nodes and edges, or only sample part of the uncertain graph [30]. This, however, will lead to information loss. (3) The last method is to build some elegant index structures, which can significantly speed up the query process without information loss, and thus has received plenty of attention recently [19, 22]. Thus, in this paper we adopt the third method for the k -NN query.

In [30], Potamias et al. studied the k -NN query on uncertain graphs by using incremental Dijkstra [29] with *Monte Carlo* (MC) sampling. In [19], Khan et al. proposed a novel index for the probabilistic reliability search problem (reliable set query), which aims to find all the nodes reachable from a given source node with probability over a user-defined threshold. Moreover, it can be adapted for answering the top- k reliability query. However, it only focuses on reliability search, and it is not clear how to support other distance functions. Recently, Maniu et al. [22] proposed a novel tree index, called ProbTree, and studied how to perform source-to-target query (STQ) using ProbTree. However, as it is mainly designed for STQ, it works poorly for k -NN queries.

In summary, none of these works can answer the k -NN queries with arbitrary probabilistic distance functions, and thus they are not general enough. Therefore, it is desirable to develop a generic k -NN query framework for any probabilistic distance functions and graphs with different probability distributions.

2.2 Probabilistic Distance Functions

All the distance functions that can be used for k -NN queries are summarized in Table 1, where $P_{s,t}(a) = \sum_{G|d_G(s,t)=a} Pr(G)$ is the probability that the shortest path distance (SPD) between two nodes s and t equals to a [30]. Given a query node s , a k -NN query can return the top- k nodes with the smallest values of d_{me} , d_{ma} , d_{ex} , and d_{er} from s , or the top- k nodes with the biggest values of d_{mp} or d_{re} from s .

The function d_{mp} measures the length of the most-probable-path (i.e., a path with the highest probability) between nodes s and t . The function d_{re} measures the probability that there exists a path between nodes s and t , which is meaningful in situations such as delivering packages in a sensor network, but it cannot deal with the cases that prefer distance rather than reliability. The function d_{me} measures the median SPD among all the possible worlds, and it can be used when the user wants to get any k -th order statistic, but its value may be infinite. The function d_{ma} computes the SPD that is the most likely to be observed when sampling a graph from \mathcal{G} . It is useful in uncertain graphs with irregular distance distributions on edges, where the value has a limited discrete domain, but it cannot deal with infinity, e.g., most values of d_{ma} will become infinite when searching in the uncertain graphs with small probability on each edge. The functions d_{ex} and d_{er} are often used to compute the expected distances, and d_{er} is better when there are infinite distances in uncertain graphs. It is worth mentioning that, there is no consensus on which distance function is the best, because different functions can be used in different applications.

Consider the example graph in Figure 3(a), where the number on an edge represents the probability that it exists and ϵ is an infinitely small number. Suppose that our goal is to compute the SPD from S to T . Then, d_{me} will return the length of the path on the top, while d_{mp} will return the length of the path on the bottom as the SPD although it may contain infinite edges in the path.

Because of space limitations, we mainly focus on *reliability* and *expected-reliable* distances in this paper, as they are the most well studied ones in recent years [19, 22, 30], but other distance functions can also be easily incorporated into our indexing framework.

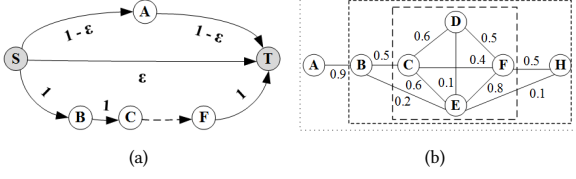


Figure 3: Examples for (a) distance functions and (b)PCD results.

3 PRELIMINARIES

In this section, we first introduce the problem we studied in Section 3.1, then discuss the MC sampling in Section 3.2, and finally present the uncertain graph decomposition (UGD) and UGD-based index in Sections 3.3 and 3.4.

3.1 Problem Definition

In line with previous studies [19, 30], we consider an uncertain graph as follows. Note that all the notations frequently used in the paper are summarized in Table 2.

DEFINITION 1 (UNCERTAIN GRAPH). An uncertain graph \mathcal{G} is a triple (V, E, p) , where V is the set of nodes, E is the set of edges, and p is the function of assigning probabilities, i.e., for any edge $e \in E$, the probability that it has a value w is $p(w|e)$, and $\sum_{w \in W} p(w|e) = 1$, where W contains all the possible values on the edges.

Let us take the graph in Figure 1(a) as an example, where each edge follows a Bernoulli distribution (e.g., for $e=(A, C)$ we have $p(0|e)=0.1$ and $p(1|e)=0.9$). For each edge, if its value $w=0$, it means that the edge does not exist; otherwise, it exists. Notice that the function p allows any kind of probability distributions (e.g., multi-valued or normal distributions) to be assigned for an edge.

According to the possible world semantics [19, 22, 30], an uncertain graph \mathcal{G} can be thought of as a probabilistic distribution, containing an exponential number of possible worlds, each having different probabilities. The probability of a possible world is defined as follows:

DEFINITION 2. Given an uncertain graph $\mathcal{G}=(V, E, p)$, the probability of observing a possible world $G=(V, E_G)$ is

$$\Pr(G) = \prod_{e \in E_G} p(i|e) \prod_{e \in E \setminus E_G} p(0|e). \quad (1)$$

For simplicity, if the edges follow Bernoulli distributions (see Figure 1(a)), the probability of observing a possible world is:

$$\Pr(G) = \prod_{e \in E_G} p(e) \prod_{e \in E \setminus E_G} (1 - p(e)). \quad (2)$$

Next, we formally define STQ and k -NN query as follows¹. We illustrate them via Example 1.

DEFINITION 3. Given an uncertain graph $\mathcal{G}(V, E, p)$, a distance function d , and two nodes s and t ($s, t \in V$), the source-to-target query (STQ) $q_d(s, t)$ aims to compute the distance between s and t based on the distance function d .

DEFINITION 4. Given an uncertain graph $\mathcal{G}(V, E, p)$, a function d , a node s ($s \in V$), and an integer $k > 0$, the k -nearest neighbors

¹For convenience, the subscript d can be omitted when the type of distance function is not required.

Table 2: Notations used in this paper.

Notation	Meaning
$\mathcal{G}(V, E, p)$	An uncertain graph
n, m	The sizes of V and E respectively
$G(V, E_G)$	A possible world from \mathcal{G}
STQ	The source-to-target query
SPD	The shortest path distance
$q(s, t)$	An STQ between two nodes s and t
$q_k(s)$	An k -NN query for a node s
$\phi(G)$	The diameter of the uncertain graph \mathcal{G}
$(\mathcal{B}, \mathcal{T})$	The tree index of an uncertain graph with the bag set \mathcal{B} and tree structure \mathcal{T}
$g(\mathcal{G}, \epsilon, \delta)$	The sampling times for \mathcal{G} with error rate ϵ and failure rate δ
$f(\mathcal{G})$	The cost function of the uncertain graph \mathcal{G}

query (k -NN) $q_k(s)$ aims to find a set of nodes C such that for any $r \in C$, the value of $d(s, r)$ is in the top- k list w.r.t. the function d .

In this paper, we will focus on the k -NN query. One naïve way to answer k -NN query is to run $(n-1)$ times of STQs like the way to answer $q_1(A)$ in Example 1. These STQs are called sub-queries of $q_k(s)$.

EXAMPLE 1. Consider the uncertain graph in Figure 1(a). (1) Let $s=A, t=B$ and $d = d_{er}$. There are 3 kinds of possible SPD values between A and B on \mathcal{G} , that is, $\Pr(\text{SPD} = 1) = 0.7, \Pr(\text{SPD} = 3) = 0.3 \times 0.9 \times 0.6 \times 0.8 = 0.1296$ (see Figure 1(b)) and $\Pr(\text{SPD} = 0) = 0.3 \times (1 - 0.9 \times 0.6 \times 0.8) = 0.1704$. Therefore, $q(A, B) = (0.7 + 0.1296 \times 3) / (1 - 0.1704) = 1.31$. (2) Let $s=A, k=1$ and $d = d_{er}$. We first calculate the three SPD values $d_{er}(A, B) = 1.31, d_{er}(A, C) = 1.07$ and $d_{er}(A, D) = 2$. Therefore, $q_1(A) = \{C\}$.

Since the number of possible worlds is exponentially large, it is impractical to enumerate all the possible worlds. To alleviate this issue, people often sample a small number of possible worlds, then perform queries on these sampled graphs, and obtain the final answer by accumulating the results in a particular manner. Next, we will introduce Monte Carlo Sampling, which is the popular way for approximate query processing in an efficient way.

3.2 Monte Carlo Sampling

To address the curse of exponentiation, the *Monte Carlo (MC) sampling* is used to sample a small number of graph instances [22, 30], and estimate the query results by performing queries on these sampled graph instances. To sample an instance graph G , we can sequentially consider each edge of \mathcal{G} and sample it as a deterministic edge in G following the probability distribution on the edge. Intuitively, if a possible world is repeatedly sampled multiple times, it should have a high probability to exist.

To achieve a theoretical estimation accuracy, the *Chernoff bound* [30] can be applied to determine the number of possible worlds needed for a k -NN query. Given an uncertain graph \mathcal{G} , a distance function d , and a pair of nodes s and t , the accuracy of estimating the value of $d(s, t)$ by MC sampling can be well guaranteed by Lemma 1:

LEMMA 1. [30] To achieve an error rate of $\epsilon > 0$ with a failure probability of $\delta > 0$, i.e., $\Pr(|d(s, t) - d'(s, t)| \geq \epsilon d(s, t)) \leq \delta$, the number of samples needed is

$$g(\mathcal{G}, s, t, \epsilon, \delta) = \max \left\{ \frac{3}{\epsilon^2 d(s, t)}, \frac{\phi(\mathcal{G})^2}{2\epsilon^2} \right\} \cdot \ln \left(\frac{2}{\delta} \right), \quad (3)$$

Table 3: UGD methods.

Method	Lossless	Time	Pros&cons
JSD [33]	Yes	Cubic	Smaller decomposition result, accurate but slow.
SPQR [14]	No	Linear	Smaller decomposition result but not lossless.
FWD [31]	Yes	Linear	Information lossless when $\mathcal{W} = 2$ with more redundancy.
LIN [22]	Yes	Linear	Compromise between SPQR and FWD.
PCD [5]	Yes	#P-hard	Layered decomposition without information loss.
PTD [16]	Yes	#P-hard	Layered decomposition without information loss.

where $d'(s, t)$ is the estimated value of $d(s, t)$, and the function $\phi(\mathcal{G}) = \max_{(s, t) \in V \times V} d(s, t)$ is the diameter of \mathcal{G} .

In practice, one usually focuses on finding the pairs with a given threshold ρ . Note that in general ρ is not too small [30], and thus we have $\frac{\phi(\mathcal{G})^2}{2\epsilon^2} \geq \frac{3}{\epsilon^2\rho}$. Therefore, the number of needed samples is:

$$g(\mathcal{G}, \epsilon, \delta) = \frac{\phi(\mathcal{G})^2}{2\epsilon^2} \ln\left(\frac{2}{\delta}\right). \quad (4)$$

3.3 Uncertain Graph Decomposition

To enable efficient k -NN queries, offline index are usually used [34, 36–38], often based on graph decomposition methods. In Table 2, we list all the uncertain graph decomposition (UGD) methods, including *junction scan* decomposition (JSD), SPQR decomposition [14], *lineage tree* decomposition (LIN) [22], *probabilistic core* decomposition (PCD) [5], and *probabilistic truss* decomposition (PTD) [16].

In [33], Na et al. proposed JSD² for dividing a deterministic graph into several partitions by finding the junction nodes. This method can also be adapted for UGD by simply regarding the probabilities of edges as their weights. SPQR [14] is named from the optimal tree obtained by decomposing the graph into a tree of tri-connected components. FWD [31] also decomposes the tree, but limits itself at bags which have at most a limited number of nodes, which is called their treewidth \mathcal{W} . LIN [22] behaves the same as FWD, but keeps more information in the bags for better query processing. SPQR can decompose the uncertain graph optimally, but during the process it will lose some information. FWD computes probabilities exactly in the bags, but can lead to decompositions that do not reduce much the graphs. LIN can both reduce the graph drastically and allow exact computation of probabilities, but comes at a high cost in space.

PCD [5] and PTD [16] are recently proposed layered uncertain graph decomposition methods. They can extract a dense part of the uncertain graph called (k, η) -core and (k, η) -truss respectively, which has higher probability to exist. This extraction can be executed iteratively. Note that even though the exact algorithm for finding (k, η) -core or (k, η) -truss is #P-hard, approximation algorithms are provided so that they can be completed in linear time [5, 16]. For example, there are three (k, η) -cores in Figure 3(b), and the smaller ones are more dense and reliable in the uncertain graph.

²Also called *partition-based road network index* in [33]. We call it JSD because it scans the junction nodes.

3.4 UGD-based Tree Index

A naïve index is to pre-compute all the pairwise distances and store them in a matrix, and then answer a query by simply looking up the matrix. This index, however, has a space complexity of $O(n^2)$, and it is not affordable if \mathcal{G} is large. Thus, it is desirable to develop more effective indexes. Recently, people often build an index based on UGD. The rationality behind is that, by decomposing \mathcal{G} into several sub-graphs, the k -NN query can be performed on a few sub-graphs, rather than the entire graph, resulting in high query efficiency.

To build an index for \mathcal{G} , a specific UGD method from Table 3 is used to decompose it into several bags, each of which contains a set of nodes of \mathcal{G} . The bags are then organized into a tree structure. Below, we present a formal definition of the tree index.

DEFINITION 5. Given an uncertain graph $\mathcal{G}(V, E, p)$, the index of \mathcal{G} is a tuple $(\mathcal{B}, \mathcal{T})$, where $\mathcal{B} = \{B_i | i = 1, 2, \dots, l\}$ is a set of l bags from the decomposition of \mathcal{G} and \mathcal{T} is a tree, such that:

- (1) $\cup_{B_i \in \mathcal{B}} B_i = V$;
- (2) For each $(u, v) \in E$, there is $B_i \in \mathcal{B}$, s.t. $u, v \in B_i$;
- (3) There is a link between B_i and B_j in \mathcal{T} if $B_i \cap B_j \neq \emptyset$.

We illustrate the index by taking FWD decomposition as an example. We first adopt FWD to compute all the bags limited by the tree width $\mathcal{W} = 2$. After that, for each pair of bags, if they share some nodes, we link them with an edge. Since only one bag serves as the root, this index is a tree structure. For example, the tree index in Figure 5 is an example from the FWD decomposition of Figure 4(a). Especially, for PCD and PTD, we only link the two bags when a bag is directly a subset of another bag, to keep the tree structure of the index. For example, if (k_1, η_1) -core $\subset (k_2, \eta_2)$ -core and (k_2, η_2) -core $\subset (k_3, \eta_3)$ -core, we only link (k_1, η_1) -core with (k_2, η_2) -core and (k_2, η_2) -core with (k_3, η_3) -core.

Recall that we have discussed six UGD methods in Section 3.3. Generally, all these methods can be adopted in the index. Due to the space limitation, in this paper we mainly focus on FWD and PCD, since most of existing k -NN queries on uncertain graphs are based on expected distance search [30] or density search [35]. For the expected distance based k -NN queries (e.g., the k -NN queries based on Expected-reliable distance [30]), FWD can be used; for the density based k -NN queries (e.g., top- k reliability query [35]), PCD can be used, since it is useful for extracting the dense components (e.g., (k, η) -core in Figure 3(b)) with high existence probabilities.

4 THE U-TREE INDEXING FRAMEWORK

Even with the sampling methods or indexes proposed in Section 3.3, it is challenging to answer k -NN queries in a large uncertain graphs. However, with an advanced index, we can speed up the searching process by generating an uncertain subgraph in much smaller size, with enough information to answer the k -NN query. Since the uncertain subgraph is smaller, query answering will be more efficient.

For example, the subgraph with essential information (Figure 4(b)) to answer the query $d_{er}(B, E)$ is much smaller compared to the whole graph in Figure 4(a). Since the reduction on nodes will lead to an exponential decrease in the sampling times based on Definition 3.2, the tree index will speed up the probabilistic searching in a dramatic way. In another aspect, a tree index has a reasonable cost compared to other indexes. It can decrease the number of distances to be computed compared to the matrix index. For example, we only need to compute 3 d_{er} values in

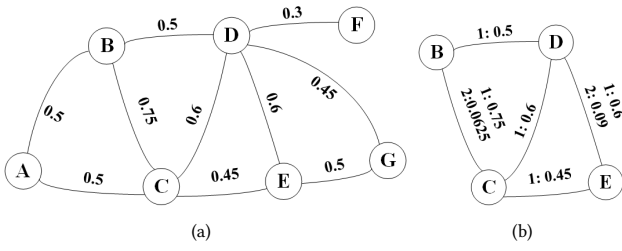


Figure 4: Tree index demonstration (a) An uncertain graph \mathcal{G} (b) $\mathcal{G}(q)$ for $q = der(B, E)$.

Figure 4(a) rather than the 21 der values for each pair of nodes in \mathcal{G} .

We will answer two questions in this section: how to build an index to efficiently answer the k -NN queries on uncertain graphs (Step A, B in Figure 2), and how to query the index when faced with a k -NN query (Step C in Figure 2). To answer the second question, we propose a novel structure and a cost evaluation model.

4.1 U-tree Index

In this section, we propose *U-tree* for k -NN search on an uncertain graph. After decomposing the uncertain graph into several bags by Definition 5, we want to dig more on the index structure.

There are two steps to build the U-tree: the basic index construction step, directly after the decomposition (Step A), and the index customization step (Step B), which will refine the index for efficient k -NN search.

Algorithm 1: Basic index construction

Input : Uncertain graph \mathcal{G}
Output : $(\mathcal{B}, \mathcal{T})$

- 1 $\mathcal{B} \leftarrow \emptyset, \mathcal{T} \leftarrow \emptyset;$
- 2 $G \leftarrow$ undirected, unweighted graph of $\mathcal{G};$
- 3 **for** $d \leftarrow 1$ to 2 **do**
- 4 **while** $degree(v) = d \ \& \ v \in G$ **do**
- 5 create bag $B;$
- 6 $V(B) \leftarrow v$ and all its neighbors;
- 7 **for all** unmarked $e \in V(B) \times V(B) \cap E(G)$ **do**
- 8 $E(B) \leftarrow E(B) \cup \{e\};$
- 9 mark $e;$
- 10 **end**
- 11 Delete v and link v 's neighbors in $G;$
- 12 $\mathcal{B} \leftarrow \mathcal{B} \cup \{B\};$
- 13 **end**
- 14 **end**
- 15 Create the root bag R with all unmarked edges;
- 16 $\mathcal{B} \leftarrow \mathcal{B} \cup \{R\};$
- 17 Organize the bags in \mathcal{B} into \mathcal{T} by their generating orders;
- 18 Add an edge between two bags in \mathcal{T} if they share nodes;
- 19 Calculate distance distributions between nodes $v \in G;$
- 20 **return** $(\mathcal{B}, \mathcal{T});$

Step A. First we adapt the STQ index from [22] into a k -NN index, representing the basic index construction method (see Algorithm 1). After the initialization (line 1 to 2), we iteratively

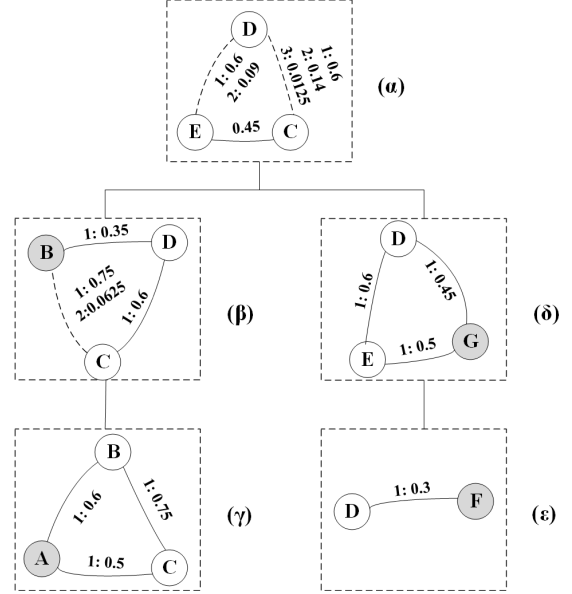


Figure 5: The basic tree index for the uncertain graph in Figure 4(a).

add the node in degree 0 or 1 into a bag with its neighbors³, as well as the edges among them to form a bag (line 3 to 14). Every time we add the node into one bag, it will be deleted from the original graph G . Then we create the root bag with the nodes left (line 15 to 16). Next, we link the bags and form a tree (line 17 to 18). This basic index is evaluated in the indexing framework named PTI in Section 5.

The index is a bottom-up structure, the root of the tree is where the query will be answered, so computations will be launched in a bottom-up way up to the root. Normally, we need several bags to answer a single sub-query; in the following, we study whether it is possible to use fewer bags to answer it after adding extra information into the index.

Note that a bag in higher level of U-tree will contain more information, because we aggregate all the information into a root bag in a bottom-up way in Algorithm 1. Consequently, only the root bag contains the complete distance distribution from the uncertain graph \mathcal{G} with respect to the nodes and edges in this bag. Then we can use the root bag itself to reduce an uncertain graph; this is not true for any other bag \mathcal{B} in the graph.

For example, node B in bag β of Figure 5 can only see the information from its child bag γ . Assume now we need to query $q_{er}(B, C)$. Even though we have known the distance distribution on the edge (B, C) in bag β , before we search the root bag α , we do not know if there exists another path in another part of the uncertain graph \mathcal{G} , that is, the subgraph of nodes $\{C, D, E, F, G\}$ in \mathcal{G} . So if the starting bag is far from the root, then we must search several bags before reaching the root bag, thus making the reduced sub-graph relatively large in size.

Step B. To reduce the bags to be scanned, we propose a novel method to customize the basic index from Step A. After generating the root bag, we make each edge contain distance information for both sides. Here both-side information means the bag can see the information of its child bags as well as the parent bags.

³We use here FWD ($\mathcal{W} = 2$) to decompose the uncertain graph into several bags. Other decomposition methods can be used with minor changes.

Then each bag will obtain a global view on \mathcal{G} . Therefore, if the sub-query $q(s, t)$ only concerns the nodes in one bag, we can use this single bag to answer $q(s, t)$, and thus the size of the uncertain subgraph will be significantly reduced.

We summarize U-tree in Algorithm 2. First we obtain the set of bags \mathcal{B} and the tree structure \mathcal{T} from the basic index (line 1). Then we initialize a queue Q and add the root bag R of \mathcal{T} into Q (line 2 to 3). Next, we iteratively pop out the head P of Q until it is not a leaf node of \mathcal{T} (line 4 to 7). For every edge that is shared by P and its child bags, we calculate the distance distribution in the corresponding child bags, with the help of the information provided by P (line 8 to 14).

Different from the basic index $(\mathcal{B}, \mathcal{T})$ from Algorithm 1, two points are developed in the index from Algorithm 2. First, each bag is embedded with more information. Second, we actually change the structure of \mathcal{T} . Since every bag now can see the whole uncertain graph and thus can serve as the root, we can terminate the index searching earlier. Because fewer bags are searched and a smaller sub-graph is generated, query time is saved when running the query on this smaller sub-graph.

Algorithm 2: U-tree construction

Input : Uncertain graph \mathcal{G}
Output : $(\mathcal{B}, \mathcal{T})$

- 1 $(\mathcal{B}, \mathcal{T}) = \text{Basic index construction}(\mathcal{G})$;
- 2 $Q \leftarrow \emptyset, P \leftarrow \text{null}$;
- 3 $Q.add(R)$;
- 4 **while** Q is not empty **do**
- 5 **while** P is null or P is the leafnode of \mathcal{T} **do**
- 6 $P \leftarrow Q.pop()$;
- 7 **end**
- 8 **for each child** S of P **do**
- 9 **if edge** e is shared by P and S **then**
- 10 Compute the distance distribution of e in S ;
- 11 $Q.add(S)$;
- 12 **end**
- 13 **end**
- 14 **end**
- 15 **return** $(\mathcal{B}, \mathcal{T})$;

For example, when answering $q_3(S)$, we need to answer the sub-query $q_{er}(A, D)$. So we start searching the U-tree in Figure 6 from bag γ and stop searching when reaching bag β . However, if we answer it by the index in Figure 5, we can only terminate the searching when reaching the root bag α . Thus, instead of generating the uncertain subgraph from $\{\alpha, \beta, \gamma\}$, a smaller uncertain subgraph is generated from $\{\beta, \gamma\}$.

Compared to the basic index, U-tree may help us utilize *graph locality* [27]. This is because the nodes, which may be queried by the users, are usually localized in some area of \mathcal{G} rather than uniformly distributed. For example, if s and t is within the same bag, a single bag is enough to answer the sub-query $q(s, t)$. From graph locality, only few bags are searched when answering each sub-query and thus the query time is significantly reduced.

The resulting extra cost is reasonable: the space complexity is still linear, and the index updating cost is only increased linearly. Using this efficient index, we can design the query algorithm for k -NN queries now.

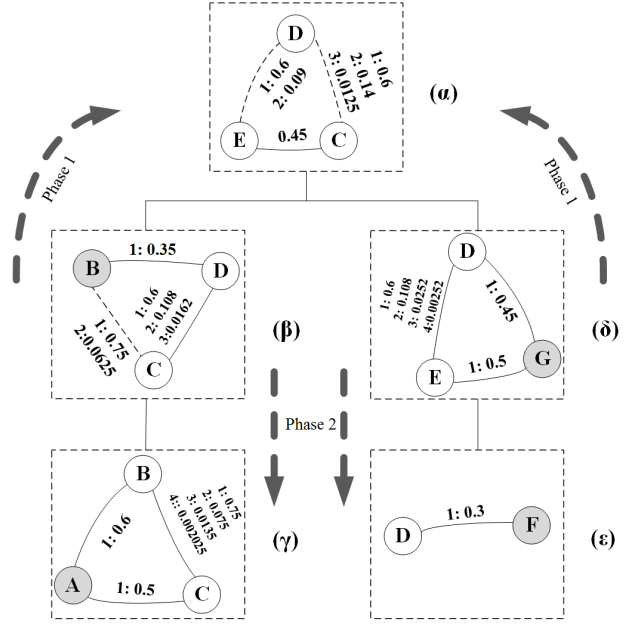


Figure 6: U-tree for the uncertain graph in Figure 4(a).

Algorithm 3: U-tree query algorithm

Input : U-tree $(\mathcal{B}, \mathcal{T})$, source node $s \in V$, integer k
Output : the k -NN set C

- 1 $C \leftarrow \emptyset$;
- 2 From s to get r on \mathcal{G} ;
- 3 Generate E-table and update the bags information;
- 4 Re-order E-table;
- 5 Initiate a queue Q and load the tuples into Q ;
- 6 Perform r -pruning;
- 7 **if** $Q = \emptyset$ **then**
- 8 **return** $NULL$
- 9 **end**
- 10 $p \leftarrow Q.pop()$;
- 11 **while** $Q \neq \emptyset$ **do**
- 12 $q \leftarrow Q.pop()$;
- 13 **if** $Cost(p \cup q) > Cost(p) + Cost(q)$ **then**
- 14 $p \leftarrow merge(p, q)$;
- 15 **else**
- 16 Generate \mathcal{G}_p and answer tuple p by sampling \mathcal{G}_p ;
- 17 $p \leftarrow q$;
- 18 **end**
- 19 Perform r -pruning in Q ;
- 20 **end**
- 21 Add the nodes with $top-k$ smallest d_{er} values into C ;
- 22 **return** C ;

4.2 Query processing by E-table

To analyze the search process on U-tree, and then develop an efficient query algorithm, we propose a novel data structure called *E-table* (from *execution table*) to keep track of the indexing querying process.

DEFINITION 6. Given a k -NN query $q_k(s)$, an *Execution-table* (*E-table*) is a collection of tuples $u = \{id, t, r, B\}$ that:

1. $u.id$ is the identifier for u .

2. $u.t$ is the target node in the sub-query $q(s, t)$ mapped to u .
3. $u.r$ is the lower bound for $d(s, t)$.
4. $u.B$ is the set of minimum bags to answer $q(s, t)$.

Note that each tuple u in the E-table corresponds to a sub-query $q(s, t)$ for the given k -NN query $q_k(s)$. A k -NN query $q_k(s)$ can be divided into $(n - 1)$ sub-queries $q(s, t_i)$ where $t_i \in V$ and $t_i \neq s$, which are actually STQs. For example, the d_{er} based NN query⁴ can be divided into $(n - 1)$ STQs and then return the node with the minimum d_{er} value. We demonstrate the use of E-table in Example 2.

EXAMPLE 2. To find the nearest neighbors for node B in Figure 4(a), we can build a E-table like Table 4, in which each row is a sub-query and each column is an attribute of the sub-query, including the id, the target node, the lower bound for d_{er} , and the bags needed to answer this sub-query. Note that for $d_{er}(s, t)$, r can be calculated from $SPD(s, t)$ since $SPD(s, t)$ can serve as the lower bound of $d_{er}(s, t)$.

Table 4: E-table for NN(B).

id	t	r	B	Sub-query
1	A	1	$\{\alpha, \beta, \gamma\}$	$d(B, A)$
2	C	1	$\{\alpha, \beta\}$	$d(B, C)$
3	D	1	$\{\alpha, \beta\}$	$d(B, D)$
4	E	2	$\{\alpha, \beta\}$	$d(B, E)$
5	F	2	$\{\alpha, \beta, \delta, \epsilon\}$	$d(B, F)$
6	G	2	$\{\alpha, \beta, \delta\}$	$d(B, G)$

Obviously, it will be very time-consuming if we run every STQ in the E-table. Alternatively, we find that a tuple in the E-table can actually answer several sub-queries. For example, in Table 4, we can find that bag α and β contain enough information to answer three sub-queries, that is, the $d_{er}(B, C)$, $d_{er}(B, D)$ and $d_{er}(B, E)$.

For a k -NN query to find the nodes t with top- k smallest $d(s, t)$, we can filter the tuples u_i whose r_i is bigger than the current maximum $d(s, t_k)$ named as d_{max} . Here t_k is the node that has been searched and added in the k -NN set to be returned. We name the k -NN set to be returned as C . It is actually the candidate set before it is returned. Note that if C is not full, the current maximum $d(s, t_k)$ is set to be $+\infty$. The filtering theory can be formalized by Lemma 2.

LEMMA 2. All the tuples t_i whose $r_i > d(s, C)$ can be safely pruned if $|C| > k$ is satisfied. Here $d(s, C) = \max_{i \in C} d(s, i)$.

The lemma can be easily proved by the fact that the k -NN solution can only be found in the subset whose r value is not bigger than the current $d_{max} = d_{er}(s, C)$. We call this pruning technique r -pruning.

Thus, our goal is to find the proper C and use it to prune the tuples in the E-table as more as possible, especially the tuple with huge amount of bags. For example, it will be nice if tuple 5 and 6 in Table 4 are all pruned at the very first stage because these two tuples will generate uncertain graphs of bigger size.

We follow two steps to generate the candidate set C . First, we rank the E-table according to r . Then, we rank the tuples according to the number of bags, that is, the tuple with fewer

⁴NN query is the k -NN query when $k = 1$.

Table 5: Executed tuples in Table 4.

id	t	r	B	sub-query	$d_{er}(s, t)$
1	A	1	$\{\alpha, \beta, \gamma\}$	$d(B, A)$	1.3 340
2	C	1	$\{\alpha, \beta\}$	$d(B, C)$	1.1631
3	D	1	$\{\alpha, \beta\}$	$d(B, D)$	1.4708

bags will be searched first even they have the same r value. It is from the instinct that the tuple u_i with fewer bags will generate a smaller uncertain sub-graph and thus may terminate in an earlier stage. We demonstrate the process in Example 3.

EXAMPLE 3. For a NN query q_1s , we load the first tuple u_1 and generate the corresponding reduced uncertain graph \mathcal{G}_1 from B_1 . Then, we sample the reduced uncertain graph \mathcal{G}_1 and get the result of $d_{er}(s, t_1)$. We add t_1 into C and update $d_{max} = d_{er}(s, t_1)$. Next, we prune the tuples whose r value is even bigger than $d_{er}(s, t_1)$. So on so forth, we load the tuple u_i which is not filtered, and generate \mathcal{G}_i from B_i to answer $d_{er}(s, t_i)$ and then launch tuple filtering and if a smaller $d_{er}(s, t_i)$ is found, update $C = t_i$ and $d_{max} = d_{er}(s, t_i)$. After searching all the tuples that are not filtered, we can return C .

4.3 Query scheduling by cost evaluation

In the steps above, we notice that the set of bags of one sub-query maybe the upper set of another sub-query, that is $B_i \subset B_j$ happens time to time. At this time, we can safely use the upper set B_j to answer u_i and u_j . For example, the uncertain graph generated from u_1 contains all the information needed to answer u_2, u_3 and u_4 . However, some sub-queries' bag sets have overlap but not belong to each other, e.g., B_1 and B_6 . Faced with this condition, we have two choices. The first choice is to combine these two tuples and generate an uncertain graph of big size to answer these two sub-queries. The second choice is to generate two small uncertain graphs and answer the two sub-queries respectively. Thus, we can choose the one with smaller cost. The cost evaluation is summarized by the function below.

DEFINITION 7. Cost Evaluation is a function f to compare the cost of combining two tuples with the cost of answering them respectively. Here each uncertain graph \mathcal{G}_i is generated by the corresponding tuple t_i , and $\mathcal{G}_{i \cup j}$ is generated by $B_i \cup B_j$. If $f(B_i \cup B_j) < f(B_i) + f(B_j)$ then we will use B_i and B_j to generate a single uncertain subgraph. Otherwise, we will generate two uncertain subgraphs respectively.

$$f(B_i \cup B_j) = g(\mathcal{G}_{i \cup j}, \epsilon, \delta) \cdot |E(\mathcal{G}_{i \cup j})| \quad (5)$$

$$f(B_i) + f(B_j) = \sum_{p=i, j} g(\mathcal{G}_p, \epsilon, \delta) \cdot |E(\mathcal{G}_p)| \quad (6)$$

If we combine all the possible tuples to generate a single uncertain graph which contains all the information to answer the k -NN query, then this reduced uncertain graph is called *Equivalent Uncertain Graph* (EUG) of \mathcal{G} for the given query. Figure 7 is the EUG for $q_1(B) = NN(B)$ on the uncertain graph in Figure 4(a). When the candidate set is big in size or the k -NN query has a large k value, the EUG cannot be reduced much. This shows another reason why we need a cost evaluation.

According to section 3.2, $\phi(\mathcal{G})$ can be roughly approximated by $(n-1)$, then the cost function can be written as $f(\mathcal{G}) = (n-1)^2 \cdot m$.

EXAMPLE 4. If there comes three tuples in the E-table, and they need the bags $\{\alpha, \beta, \gamma\}$, $\{\alpha, \delta, \epsilon\}$ and $\{\alpha, \beta, \delta\}$ respectively (see Figure 8). Thus we get three reduced uncertain graph $\mathcal{G}_1, \mathcal{G}_2$ and \mathcal{G}_3 .

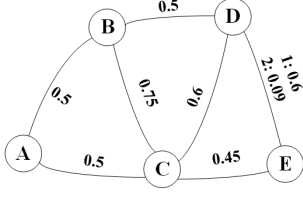


Figure 7: The EUG for Figure 4(a) considering NN(B).

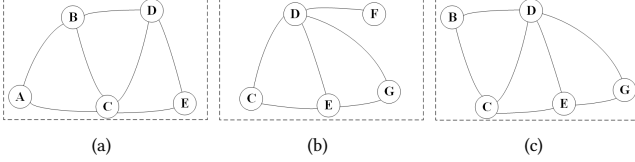


Figure 8: The uncertain graph generated from bags (a) $\{\alpha, \beta, \gamma\}$, (b) $\{\alpha, \delta, \epsilon\}$ and (c) $\{\alpha, \beta, \delta\}$.

We can merge these reduced graphs to answer multiple tuples or answer them one by one. There are 5 ways to answer these tuples and the costs are listed below.

$$\begin{aligned}
 f(\mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3) &= f(\mathcal{G}_1) + f(\mathcal{G}_2) + f(\mathcal{G}_3) = 4^2 \times (7 + 6 + 7) = 320 \\
 f(\mathcal{G}_1 \cup \mathcal{G}_2, \mathcal{G}_3) &= f(\mathcal{G}_1 \cup \mathcal{G}_2) + f(\mathcal{G}_3) = 6^2 \times 10 + 4^2 \times 7 = 472 \\
 f(\mathcal{G}_1, \mathcal{G}_2 \cup \mathcal{G}_3) &= f(\mathcal{G}_1) + f(\mathcal{G}_2 \cup \mathcal{G}_3) = 4^2 \times 7 + 5^2 \times 8 = 312 \\
 f(\mathcal{G}_2, \mathcal{G}_1 \cup \mathcal{G}_3) &= f(\mathcal{G}_2) + f(\mathcal{G}_1 \cup \mathcal{G}_3) = 4^2 \times 6 + 5^2 \times 9 = 321 \\
 f(\mathcal{G}_1 \cup \mathcal{G}_2 \cup \mathcal{G}_3) &= f(\mathcal{G}_1 \cup \mathcal{G}_2 \cup \mathcal{G}_3) = 6^2 \times 10 = 360
 \end{aligned}$$

Here we see the optimal solution for this E-table is not to answer each tuple respectively or to generate an EUG, but only to merge \mathcal{G}_2 and \mathcal{G}_3 . Therefore, we form Algorithm 3 to find the local optimal solution of the minimum cost. We find that the local optimal solution works well; on the other hand, finding the global optimal solution is relatively cumbersome.

Given the initial uncertain graph $\mathcal{G}_0 = (V_0, E_0, p)$ and two reduced uncertain graph $\mathcal{G}_1 = (V_1, E_1, p)$, $\mathcal{G}_2 = (V_2, E_2, p)$, the function $merge(\mathcal{G}_1, \mathcal{G}_2)$ generates an uncertain graph $\mathcal{G}_{1,2} = (V_1 \cup V_2, E, p)$ where $E = (V_1 \cup V_2)^2 \cap E_0$. The function pop will pop up the queue's head and delete it from the queue.

Discussion. We notice that the basic index has a fixed root, that is, every time when the index is queried, we need to reach the root. But every bag in the U-tree can serve as the root, meaning the index retrieval can be done in any part of the tree. So when the query node is far away from the root, the query time from the basic index will become too high, because many bags are required and the reduced subgraph will not decrease much in size. U-tree overcomes this drawback but the index construction time is increased. Therefore, we study whether there is any trade-off and balance between the two indexes, and then we can enjoy both the quick indexing and efficient querying. We answer this question by graph locality. With the help of the history queries, we can obtain the popular nodes which may be frequently queried [18]. And with the help of graph locality, most k -NN queries will be around these nodes. Then we can build multiple basic indexes rooted on each popular node. Every time a query comes, it can be assigned to an index with the smallest cost. In this way, the cost of index updating is reduced and the query is accelerated.

5 EXPERIMENTAL EVALUATION

We now present the experimental results. We first describe the datasets in Section 5.1, then introduce the competitors in Section 5.2, and finally report the experimental results on indexing cost, query time, time proportion, accuracy and generality in Sections 5.3, 5.4, 5.5, 5.6, and 5.7 respectively.

5.1 Datasets

5.1.1 Real-world Graphs. We use six real-world datasets. Table 6 reports their statistics. All datasets are transformed into undirected uncertain graphs with meaningful probability distribution on each edge. Note that PPI, DBLP and FBN have Bernoulli distributions on each edge, whereas BJT has no limitation on the type of the distribution on each edge, and they do not follow any standard distributions [8].

For each uncertain graph, we calculate the diameter to better show the size and density of the dataset. To calculate the diameter, we ignore the probabilities and treat each uncertain graph as an unweighted graph, and then find the largest SPD. For the dataset with multiple connected components, we report the one with the biggest diameter. We evaluate scalability of the indexing framework by testing its performance when faced with datasets with different size (see Table 6) and different density (See Table 7).

PPI⁵ We have two protein-protein interaction (PPI) networks (PPIC, PPIK) [25]. We use these networks in which a node denotes a protein and an edge denotes a possible interaction encoded with a specific probability calculated from biology experiments, meaning the existence confidence of this edge. *Collins's* network PPIC is the core dataset which is believed to have strong interactions among the nodes [7]. *Krogan's* network PPIK, however, contains less reliable interactions and thus is bigger in size [20].

DBLP⁶ It is a subset of the co-authorship network. The nodes denote the authors and if two authors have coauthored a paper, there will be an edge between them. We follow the popular way of generating co-authorship probability between two authors [19]. If two authors coauthored c times, the probability is $1 - e^{-c/\mu}$ on the corresponding edge where smaller μ means smaller probabilities in general. Following the trend [19], we adopt $\mu = 5$ which makes the most edges with an existing probability around 0.05.

FBN⁷ The dataset called Facebook-like Social Network is from an online community for students at University of California, Irvine [26]. It includes the users who sent or received at least one message. Each node denotes a user and if there are messages between two users, there will be an edge. The probability on an edge means the probability of the corresponding users having an interaction. The method to generate the probability is same as the method in DBLP but we set $\mu = 2$ to make the most edges with a probability around 0.4.

BJT⁸ The dataset is generated from mapping 15 million Beijing Taxi trajectories [39, 40] on a Beijing map with about 3 million road segments (BJT) and 1 million road segments (BJTA, only the arterial roads) respectively [33]. The probability distribution on an edge means the speed distribution of the vehicles on the corresponding road segment. It can be generated by analyzing the trajectories [8]. Note that the diameters of BJT and BJTA is obviously bigger than the other datasets, since long roads (tens

⁵<http://www.nature.com.eproxy1.lib.hku.hk/nmeth/journal/v9/n5/full/nmeth.1938.html>

⁶<http://www.informatik.uni-trier.de/~ley/db/>

⁷<https://toreopsahl.com/datasets/>

⁸<https://www.microsoft.com/en-us/research/publication/t-drive-trajectory-data-sample/>

Table 6: Statistics of Real-world Datasets.

Graph	#Nodes	#Edges	Diameter	Degree
PPIC	1,622	9,074	15	11.2
PPIK	3,672	14,317	10	7.8
FBN	22,016	58,595	10	5.3
DBLP	317,080	1,033,668	23	6.6
BJTA	426,196	946,434	3,851	4.4
BJT	1,285,215	2,690,296	10,529	4.2

Table 7: Statistics of Synthetic Datasets.

Graph	#Nodes	#Edges	Diameter
S ₁	1,000,000	1,115,373	15
S ₂	1,000,000	1,672,561	15
S ₃	1,000,000	2,433,185	15
S ₄	1,000,000	3,128,529	15
S ₅	1,000,000	3,728,130	15
S ₆	1,000,000	4,213,833	15

of kilometers) will be separated into many road segments (ten meters).

5.1.2 Synthetic Graphs. Given an unweighted graph, algorithms from [4] can generate a graph with disturbances on the edges and calculate a probability of existence for each edge. The algorithm from [23] can encrypt a given weighted graph into an uncertain graph encoded with a probability on each edge.

Here we encrypt a synthetic graph generated from the *Barabási-Albert model* [3], which is a widely used model to simulate real graphs. To vary the density of the graph, we set the diameter as 15 but vary the number of the edges for each synthetic graph. Then the unweighted graphs with different size are encrypted by $(20, 10^{-3})$ -obfuscation [4]. We use these graphs to test the scalability of the indexes.

5.2 Competitors

In our experiments, we try to find the k -nearest neighbors for 100 randomly generated query nodes. We use three competitors: Incremental k -NN (IKNN) from [30], RQ -tree reliability search (RQRS) from [19], and the basic index (see Algorithm 1) adapted from *ProbTree* indexing framework (PTI) [22].

We adapt PTI into a single-source version to deal with k -NN queries. They have been discussed in the related works. IKNN is popular in finding k -NN in the uncertain graphs, and RQRS is the state-of-the-art probabilistic k -NN index based on reliability search. PTI is the current indexing framework which can adopt different decomposition methods and distance functions.

We implement the whole U-tree indexing framework in Java, and run experiments on a machine having a 4-core Intel i5-3570 3.40GHz processor and 16GB of memory, with Ubuntu installed.

5.3 Evaluation on Indexing Cost

Here we evaluate the space cost and the index construction time of U-tree, PTI and RQRS.

The space cost is composed by the tree structure and the distance distribution stored in the bags. From Figure 9, we see that the U-tree index needs more memory since it stores more distance distributions into each bag, and RQRS needs more memory because it stores the whole node set at each level of the tree index. Compared with the query efficiency improvement in Figure 13,

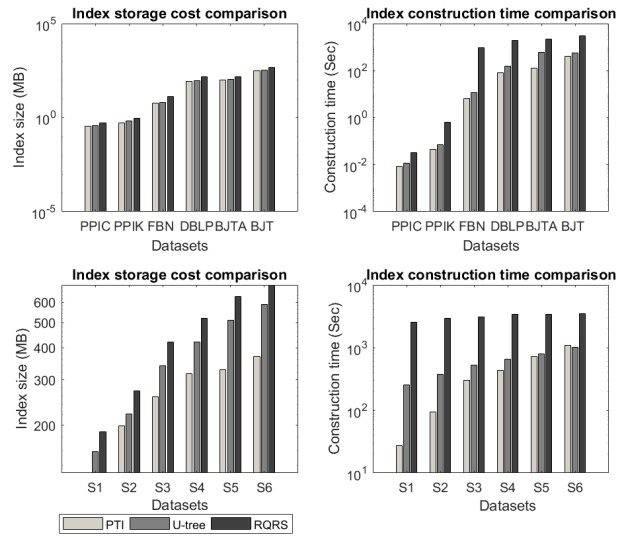


Figure 9: Scalability Evaluation on index size and index construction Time on real-world datasets (top) and synthetic datasets (bottom).

we find that the extra memory cost in U-tree index is reasonable, because the space cost rises 23% but the query time is 75% faster on average.

We compare the time cost to construct each index in Figure 9. To avoid affecting the comparison, here we only evaluate the time to construct the tree index and the uncertain graphs are assumed to be decomposed beforehand. Note that RQRS needs to recursively perform balanced bi-partition clustering, which can also be seen as a decomposition method.

To test the scalability of U-tree, we use the synthetic graph with the graph anonymization technique. We vary the size of the synthetic graph (see Table 7) and compare the index construction time and index storage in Figure 9. *Semilog* is used on the y -axis because of the big difference of the index size and construction time when the number of nodes sharply changes. From the figure, we find the index construction cost of U-tree is linear to the number of nodes in the uncertain graph. This shows good scalability over uncertain graphs of different size.

5.4 Evaluation on Query Time

Here we evaluate the query time on six real-world networks and six synthetic datasets. We use the popular probabilistic k -NN algorithm IKNN [30] to show the efficiency of the query time without index acceleration, and the ability of U-tree, PTI and RQRS to speed up the searching. This comparison makes sense because they all need to find the probability distribution among nodes, which is the most expensive part in the algorithms.

In Figure 13, we vary the k value to see the changes of each curve. For each k value, we randomly pick 100 nodes as the source nodes to initiate 100 probabilistic k -NN queries, and calculate the average query time as the value on y -axis. We use *semilog* on the query time because of the wide range between different methods.

From Figure 13, U-tree is superior to the other methods, and the index based methods have a much smaller query time than IKNN (about two magnitudes). However, the curve of RQRS and that of PTI index twist together. This might be because both RQRS

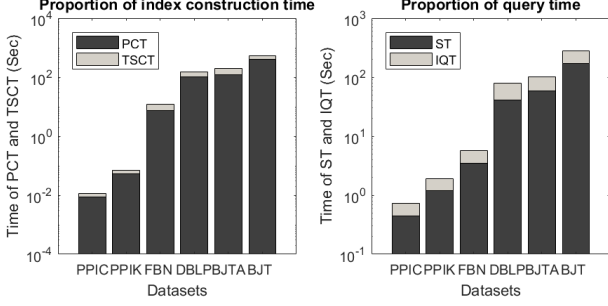


Figure 10: Proportion of (a) probability computation time out of index construction time (b) index retrieval time out of query time.

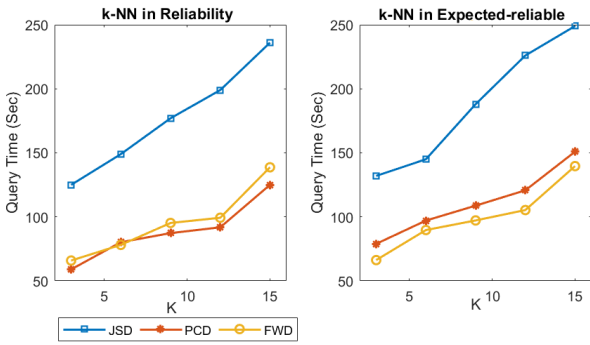


Figure 11: Comparison of (a) k -NN query in d_{re} and (b) k -NN query in d_{er} on the U-tree from JSD, PCD and FWD.

and PTI construct a tree whose height $h \in [10, 20]$ and they all have to search on the tree from the leaves until reaching the big root. These similarities make their retrieval time in the same level. Also, the figures show good scalability of U-tree despite of the dataset size and density.

5.5 Evaluation on Time Proportion

From Figure 10 we see that the Probability Computation Time (PCT) takes the most part of the index construction time and the Sampling Time (ST) takes the most part of the query time. Note that the query time minus ST is the Index Query Time (IQT) and the index construction time minus PCT is the Tree Structure Construction Time (TSCT).

It makes sense because in the index construction period, to calculate the distance distributions is required in both phase one and phase two which asks for sampling the corresponding subgraphs. Also, when we run the queries on the index, the bags containing essential information are collected and then sample the reduced uncertain subgraph.

Since bigger proportion of TSCT means more complex index structure, e.g., a bigger tree height, and thus requires more time to answer a k -NN query. Also, IQT is the time to be spent on searching the index. The bigger proportion of an IQT, the bigger probability that more bags are required to generate the uncertain subgraph, thus making the query time unbearable. From Figure 10, U-tree performs good when searching the index and answering the k -NN queries.

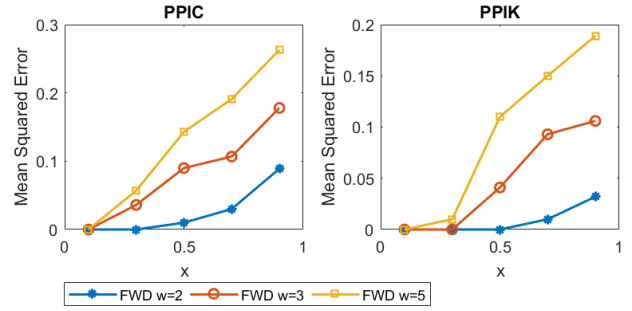


Figure 12: Indexing framework accuracy evaluation.

5.6 Evaluation on Accuracy

In U-tree indexing framework, there are two parts that may cause precision loss. The first part is the choice of decomposition methods. If a decomposition method which may lose information is used, e.g., FWD with $\mathcal{W} > 2$, the uncertain subgraph to be generated from searching the index will become not accurate, thus making error in k -NN results. The second part is the sampling method. Since sampling is a technique for approximate query processing, it will lose information while accelerating the querying process. In general, less sampling times will lead to bigger error in the k -NN results.

Here we vary the FWD tree width \mathcal{W} and construct the corresponding U-tree respectively. Then, we run k -NN query on these U-trees with different pair of parameters (ϵ, δ) in the sampling. We set $(\epsilon, \delta) = (x, x)$ and vary x from 0.1 to 0.9.

The k -NN results are collected and compared with the k -NN result from a baseline which simply runs big number of samplings times to find the k -nearest neighbors on uncertain graphs. We run the experiment on PPIC and PPIK and d_{er} is used. Here k is set to 100 and the Mean Squared Error is reported.

From Figure 12, we see that the Mean Squared Error is generally small even when the sampling parameters are setting in a bad way, e.g., ϵ and δ are very near to 1. Especially, with parameters below 0.25, we can achieve high accurate k -NN results even with decomposition methods that may lose information.

5.7 Evaluation on Framework Generality

To show the generality of the indexing framework, we will use another distance function and another decomposition method to construct U-tree. We run two different k -NN queries on each U-tree, i.e., the k -NN query which asks for the set of nodes with top- k biggest reliabilities d_{re} from the query node, and the k -NN query which asks for the k -nearest neighbors in Expected-reliable distance d_{er} .

We use three different decomposition methods to construct U-tree, i.e., JSD which can find the nodes and edges to divide the graph into several partitions [33], PCD (see Figure 3(b)), and FWD where we set the tree width $\mathcal{W} = 2$ [22].

We run this evaluation on DBLP. The results are collected in Figure 11. We can see that the U-tree with PCD and the U-tree with FWD are similar in performance, and PCD has a little advantage over FWD for k -NN in d_{re} while FWD is better for k -NN in d_{er} . The above two U-trees have significant advantage over the U-tree with JSD, since JSD performs poor in capturing hierarchical structure of an uncertain graph.

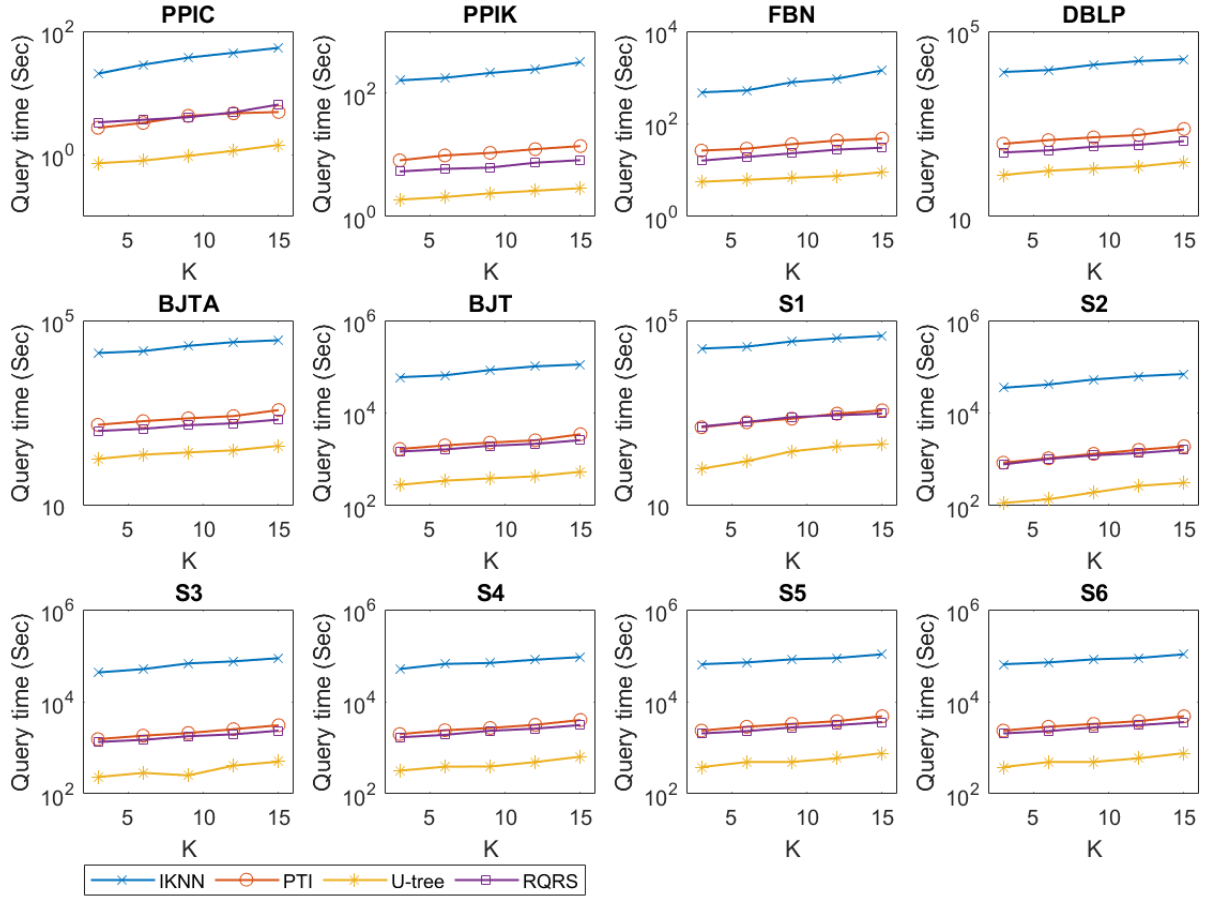


Figure 13: Query efficiency comparison.

6 CONCLUSION

In this paper, we examine the k -NN query on uncertain graphs. We first propose a generic index which can be built with several uncertain graph decomposition methods. Then, based on this index, we develop an efficient k -NN query algorithm, which supports various probabilistic distance functions. Finally, we evaluate the proposed indexing frameworks on both real and synthetic large uncertain graphs. The experimental results show that our index is effective and scalable to large graphs, and our query algorithm is very efficient.

In the future, we will study how to efficiently maintain the index for dynamic graphs, in which the nodes and edges are inserted and deleted frequently. Another interesting direction is to study how to extend the indexing framework for answering k -NN queries on a distributed platform. The research on the insight of the uncertain graph decomposition will be valuable, since a better decomposition method will largely benefit the index performance. We will also perform more experimental evaluations on other real uncertain graphs.

A APPENDIX

In this appendix, we present the algorithm listing of the competitors from the research literature.

Algorithm 4: Incremental k -NN (IKNN)

Input : uncertain graph $\mathcal{G} = (V, E, P, W)$, query node $s \in V$, sampling times r , distance increment γ , k
Output : k -NN result T_k

- 1 $T_k \leftarrow \emptyset, D \leftarrow 0$;
- 2 Initiate r executions of *Dijkstra* from s ;
- 3 **while** $|T_k| < k$ **do**
- 4 $D \leftarrow D + \gamma$;
- 5 **for** $i \leftarrow 1$ **to** r **do**
- 6 Continue visiting nodes until reaching D by *Dijkstra*;
- 7 **for each node** $t \in V$ **visited do**
- 8 | Update the distance distribution and get d_{er} ;
- 9 **end**
- 10 **end**
- 11 **for node** $t \notin T_k$ **do**
- 12 | **if** $d_{er}(s, t) < D$ **then**
- 13 | $T_k \leftarrow T_k \cup \{t\}$
- 14 | **end**
- 15 **end**
- 16 **end**
- 17 **return** T_k ;

Algorithm 5: *RQ*-tree reliability search (RQRS)

Input : uncertain graph $\mathcal{G} = (V, E, P)$, query node $s \in V$,
sampling times r, k

Output : k -NN result T_k

- 1 $T_k \leftarrow \emptyset$;
 - 2 Initiate *RQ*-tree based on binary clustering;
 - 3 Compute the upper bound U_{out} for each cluster C ;
 - 4 Generate the candidate set C^* ;
 - 5 Travel the clusters in *RQ*-tree in bottom-up until
 $C^*({s}, \eta) = \arg \max_{\{S\} \subseteq C, U_{out}(\{s\}, C) < \eta} U_{out}(\{s\}, C)$;
 - 6 Get the reduced graph G' by C^* ;
 - 7 Initiate r executions of *DFS* on G' from s ;
 - 8 Calculate the reliability and select the biggest *top-k* as T_k ;
 - 9 return T_k ;
-

Algorithm 6: *ProbTree* indexing (PTI) (\mathcal{G})

Input : uncertain graph $\mathcal{G} = (V, E, P)$, query node $s \in V$,
sampling times r, k

Output : k -NN result T_k

- 1 # Index construction:
 - 2 Run Algorithm 1 to build the *FWD* tree with $\mathcal{W} = 2$;
 - 3 # Information Retrieval:
 - 4 $T_k \leftarrow \emptyset$;
 - 5 **for** $v \in V$ **do**
 - 6 | Calculate $d_{er}(s, v)$ with the help of the *FWD* tree;
 - 7 **end**
 - 8 Put the k nodes with the *top-k* smallest $d_{er}(s, v)$ in T_k ;
 - 9 return T_k ;
-

REFERENCES

- [1] Eytan Adar and Christopher Re. 2007. Managing uncertainty in social networks. *IEEE Data Eng. Bull.* 30, 2 (2007), 15–22.
- [2] Saurabh Asthana, Oliver D King, Francis D Gibbons, and Frederick P Roth. 2004. Predicting protein complex membership using probabilistic network reliability. *Genome research* 14, 6 (2004), 1170–1175.
- [3] Albert-László Barabási and Réka Albert. 1999. Emergence of scaling in random networks. *science* 286, 5439 (1999), 509–512.
- [4] Paolo Boldi, Francesco Bonchi, Aristides Gionis, and Tamir Tassa. 2012. Injecting uncertainty in graphs for identity obfuscation. *Proceedings of the VLDB Endowment* 5, 11 (2012), 1376–1387.
- [5] Francesco Bonchi, Francesco Gullo, Andreas Kaltenbrunner, and Yana Volkovich. 2014. Core decomposition of uncertain graphs. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 1316–1325.
- [6] Reynold Cheng, Yixiang Fang, and Matthias Renz. 2014. *Data Classification: Algorithms and Applications*. Chapman & Hall CRC Data Mining and Knowledge Discovery Series, New York, USA, Chapter Uncertain Data Classification.
- [7] Sean R Collins, Patrick Kemmerer, Xue-Chu Zhao, Jack F Greenblatt, Forrest Spencer, Frank CP Holstege, Jonathan S Weissman, and Nevan J Krogan. 2007. Toward a comprehensive atlas of the physical interactome of *Saccharomyces cerevisiae*. *Molecular & Cellular Proteomics* 6, 3 (2007), 439–450.
- [8] Jian Dai, Bin Yang, Chenjuan Guo, Christian S Jensen, and Jilin Hu. 2016. Path cost distribution estimation using trajectory data. *Proceedings of the VLDB Endowment* 10, 3 (2016), 85–96.
- [9] Yixiang Fang, Reynold Cheng, Xiaodong Li, Siqiang Luo, and Jiafeng Hu. 2017. Effective community search over large spatial graphs. *Proceedings of the VLDB Endowment* 10, 6 (2017), 709–720.
- [10] Yixiang Fang, Reynold Cheng, Siqiang Luo, and Jiafeng Hu. 2016. Effective community search for large attributed graphs. *Proceedings of the VLDB Endowment* 9, 12 (2016), 1233–1244.
- [11] Yixiang Fang, Reynold Cheng, Wenbin Tang, Silviu Maniu, and Xuan Yang. 2016. Scalable algorithms for nearest-neighbor joins on big trajectory data. *IEEE Transactions on Knowledge and Data Engineering* 28, 3 (2016), 785–800.
- [12] Yixiang Fang, Haijun Zhang, Yunming Ye, and Xutao Li. 2014. Detecting hot topics from Twitter: A multiview approach. *Journal of Information Science* 40, 5 (2014), 578–593.
- [13] Joy Ghosh, Hung Q Ngo, Seokhoon Yoon, and Chunming Qiao. 2007. On a routing problem within probabilistic graphs and its application to intermittently connected networks. In *INFOCOM 2007. 26th IEEE International Conference on Computer Communications*. IEEE, 1721–1729.
- [14] Carsten Gutwenger and Petra Mutzel. 2001. A linear time implementation of SPQR-trees. In *Graph Drawing*. Springer, 77–90.
- [15] Jiafeng Hu, Reynold Cheng, Zhipeng Huang, Yixiang Fang, and Siqiang Luo. 2017. On Embedding Uncertain Graphs. In *26th ACM Conf. on Information and Knowledge Management (ACM CIKM 2017)*.
- [16] Xin Huang, Wei Lu, and Laks VS Lakshmanan. 2016. Truss decomposition of probabilistic graphs: Semantics and algorithms. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 77–90.
- [17] Zhipeng Huang, Yudian Zheng, Reynold Cheng, Yizhou Sun, Nikos Mamoulis, and Xiang Li. 2016. Meta structure: Computing relevance in large heterogeneous information networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 1595–1604.
- [18] Theodore Johnson and Dennis Shasha. 1994. X3: A low overhead high performance buffer management replacement algorithm. In *Proceedings of VLDB*.
- [19] Arijit Khan, Francesco Bonchi, Aristides Gionis, and Francesco Gullo. 2014. Fast Reliability Search in Uncertain Graphs. In *EDBT*. 535–546.
- [20] Nevan J Krogan, Gerard Cagney, Haiyuan Yu, Gouqing Zhong, Xinghua Guo, Alexandr Ignatchenko, Joyce Li, Shuyue Pu, Nira Datta, Aaron P Tikuisis, et al. 2006. Global landscape of protein complexes in the yeast *Saccharomyces cerevisiae*. *Nature* 440, 7084 (2006), 637–643.
- [21] Xiang Li, Yao Wu, Martin Ester, Ben Kao, Xin Wang, and Yudian Zheng. 2017. Semi-supervised clustering in attributed heterogeneous information networks. In *Proceedings of the 26th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee.
- [22] Silviu Maniu, Reynold Cheng, and Pierre Senellart. 2017. An Indexing Framework for Queries on Probabilistic Graphs. *ACM Transactions on Database Systems (TODS)* 42, 2 (2017), 13.
- [23] Xianrui Meng, Seny Kamara, Kobbi Nissim, and George Kollios. 2015. GRECS: graph encryption for approximate shortest distance queries. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*.
- [24] Naoki Nariai, Eric Kolaczyk, and Simon Kasif. 2007. Probabilistic protein function prediction from heterogeneous genome-wide data. *PLoS One* (2007).
- [25] Tamás Nepusz, Haiyuan Yu, and Alberto Paccanaro. 2012. Detecting overlapping protein complexes in protein-protein interaction networks. *Nature methods* 9, 5 (2012), 471–472.
- [26] Tore Opsahl and Pietro Panzarasa. 2009. Clustering in weighted networks. *Social networks* 31, 2 (2009), 155–163.
- [27] Dominic Pacher, Robert Binna, and Günther Specht. 2011. Data Locality in Graph Databases through N-Body Simulation. In *Grundlagen von Datenbanken*. Citeseer, 85–90.
- [28] Panos Parchas, Francesco Gullo, Dimitris Papadias, and Francesco Bonchi. 2014. The pursuit of a good possible world: extracting representative instances of uncertain graphs. In *Proceedings of the 2014 ACM SIGMOD international conference on management of data*. ACM, 967–978.
- [29] Sriram Pemmaraju and Steven S Skiena. 2003. *Computational Discrete Mathematics: Combinatorics and Graph Theory with Mathematica*. Cambridge university press.
- [30] Michalis Potamias, Francesco Bonchi, Aristides Gionis, and George Kollios. 2010. K-nearest neighbors in uncertain graphs. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 997–1008.
- [31] Neil Robertson and Paul D Seymour. 1984. Graph minors. III. Planar tree-width. *Journal of Combinatorial Theory, Series B* 36, 1 (1984), 49–64.
- [32] Petteri Sevon, Lauri Eronen, Petteri Hintanen, Kimmo Kulovesi, and Hannu Toivonen. 2006. Link discovery in graphs derived from biological databases. In *International Workshop on Data Integration in the Life Sciences*. Springer.
- [33] Na Ta, Guoliang Li, Yongqing Xie, Changqi Li, Shuang Hao, and Jianhua Feng. 2017. Signature-based trajectory similarity join. *IEEE Transactions on Knowledge and Data Engineering* 29, 4 (2017), 870–883.
- [34] Silke Trißl and Ulf Leser. 2007. Fast and practical indexing and querying of very large graphs. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. ACM, 845–856.
- [35] Leslie G Valiant. 1979. The complexity of enumeration and reliability problems. *SIAM J. Comput.* 8, 3 (1979), 410–421.
- [36] Haixun Wang, Hao He, Jun Yang, Philip S Yu, and Jeffrey Xu Yu. 2006. Dual labeling: Answering graph reachability queries in constant time. In *Proceedings of the 22nd International Conference on Data Engineering*. IEEE, 75–75.
- [37] Fang Wei. 2010. TED: efficient shortest path query answering on graphs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 99–110.
- [38] Yanghua Xiao, Wentao Wu, Jian Pei, Wei Wang, and Zhenyong He. 2009. Efficiently indexing shortest paths by exploiting symmetry in graphs. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*. ACM, 493–504.
- [39] Jing Yuan, Yu Zheng, Xing Xie, and Guangzhong Sun. 2011. Driving with knowledge from the physical world. In *Proceedings of the 17th SIGKDD international conference on Knowledge discovery and data mining*. ACM, 316–324.
- [40] Jing Yuan, Yu Zheng, Chengyang Zhang, Wenlei Xie, Xing Xie, Guangzhong Sun, and Yan Huang. 2010. T-drive: driving directions based on taxi trajectories. In *Proceedings of the 18th SIGSPATIAL international conference on advances in geographic information systems*. ACM, 99–108.

MatchCatcher: A Debugger for Blocking in Entity Matching

Han Li¹, Pradap Konda¹, Paul Suganthan G.C.¹, AnHai Doan¹,
Benjamin Snyder², Youngchoon Park³, Ganesh Krishnan⁴, Rohit Deep⁴, Vijay Raghavendra⁴

¹University of Wisconsin-Madison, ²Amazon, ³Johnson Controls, ⁴@WalmartLabs

ABSTRACT

Blocking is a fundamental step in entity matching (EM). Much work has examined the design and runtime of blockers. However, very little if any work has examined the problem of debugging blocking accuracy. In practice, blockers' accuracy can vary drastically, and using an accurate blocker is critical for many EM applications. To address this problem, we describe the MatchCatcher solution. Given two tables to be matched and a blocker, MatchCatcher finds matches killed off by the blocker, so that the user can examine these matches to understand how well the blocker does accuracy-wise and what can be done to improve its accuracy. We show how to quickly find such matches using string similarity joins, iterative user engagement, rank aggregation, and active/online learning. Extensive experiments show that MatchCatcher is highly effective in helping users develop blockers, can help improve accuracy of even the best blockers manually created or automatically learned. MatchCatcher has been open sourced and used by 300+ students in data science class projects and 7 teams at 6 organizations.

1 INTRODUCTION

Entity matching (EM) finds data instances referring to the same real-world entity [6, 14], such as tuples (Dave Smith, San Francisco, CA) and (David Smith, S.F., CA). This problem is critical for many Big Data and data science applications.

When doing EM, we often must perform blocking. Consider for example matching two tables A and B . Real-world tables often have hundreds of thousands, or millions, of tuples. Trying to match all tuple pairs in $A \times B$ is practically infeasible. So we often perform a step called *blocking* which uses domain heuristics to quickly drop many pairs judged obviously non-matched (e.g., person tuples that do not have the same state). The next step, called *matching*, matches the remaining pairs, using rule- or learning-based techniques. Blocking can greatly reduce the number of pairs considered in the matching step, drastically reducing the total EM time. As a result, virtually all real-world EM applications use blocking.

Numerous blocking methods have been developed [6]. For example, *hash blocking* drops all tuple pairs that do not have the same hash value, using a predefined hash function. This method is popular because it is easy to understand and fast. Other methods include sorted neighborhood, overlap, phonetic, rule-based, etc. (see Section 2).

Given two tables A and B to match, we often want a blocker Q that is *fast*, *selective*, and *accurate*. "Fastness" is measured by the time to apply Q to A and B to produce a set of tuple pairs C . "Selectivity" is typically measured as the ratio $|C|/|A \times B|$. "Accuracy" is typically measured as the fraction of *true matches* surviving blocker Q , i.e., $|M \cap C|/|M|$, where M is the set of

(unknown) true matches in $A \times B$. As such, it is also referred to as *recall*.

In practice, blockers can vary drastically in recall, and using a blocker with high recall is critical for many EM applications (see Section 2). Yet today there is still no good way to develop such blockers. For example, given the popularity of hash blockers, suppose we have decided to use a hash blocker Q on two tables. While fast, Q may have low recall if the attribute values to be hashed are dirty, misspelt, missing, or have many natural variations (e.g., "New York", "NY", "NYC"). A common way to address this problem is to use multiple hash blockers and take the union of their outputs, to maximize recall. However, even in this case, the recall can still be quite low. For instance, a recent work [8] describes two real-world datasets where extensive effort at combining hash blockers achieves only 38.8% and 72.6% recall. Such low recalls are simply unacceptable for many EM applications. To improve recall, we can revise the current hash blockers, replace some of them, or adding more blockers (of the non-hash types). *To do any of these, however, we need a way to understand whether the current blocker has low recall, and if so, then what the possible problems are, so that we can improve it.*

The MatchCatcher Solution: In this paper we take the first step toward solving the above problems. We describe MatchCatcher, a solution to debug blocker accuracy. Given two tables A and B to be matched and a blocker Q , MatchCatcher attempts to find matches that are "killed off" by Q , i.e., those that do not survive the blocking step. We can examine these matches to see if they are indeed true matches, and if so, then why they get killed off by Q . This tells us whether Q has low recall, and if so, then how to improve it. The following example illustrates our solution:

Example 1.1. Consider matching tables A and B in Figure 1.a. Suppose a user U begins by creating a blocker Q_1 that keeps only tuple pairs sharing the same value for "City". Figure 1.b shows this blocker as $Q_1: a.City = b.City$. (This is attribute-equivalence blocking, a special type of hash blocking.) Applying Q_1 to A and B produces a set of tuple pairs C_1 (see Figure 1.b).

User U wants to know if blocker Q_1 kills off too many true matches. To answer this, U applies MatchCatcher, which operates in iterations. In the first iteration, MatchCatcher shows the user n tuple pairs judged most likely to be matches killed off by Q_1 . These pairs are listed on Figure 1.b, under "Debugger Output, Iter 1" (here $n = 3$).

User U finds that the first two pairs, (a_1, b_1) and (a_3, b_2) , are indeed true matches (shown in red color on the figure). A closer examination reveals that they do not survive blocking because their "City" values do not match due to misspellings and abbreviation, e.g., "Atlanta" vs. "Atlanta", "New York" vs. "NY".

Next, U wants to know if there are any more true matches. Toward this goal, U flags the true matches in the first iteration (i.e., the above two pairs). MatchCatcher uses this feedback to find the next n pairs judged most likely to be killed-off matches, then shows those pairs in the second iteration (see Figure 1.b, under "Iter 2").

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

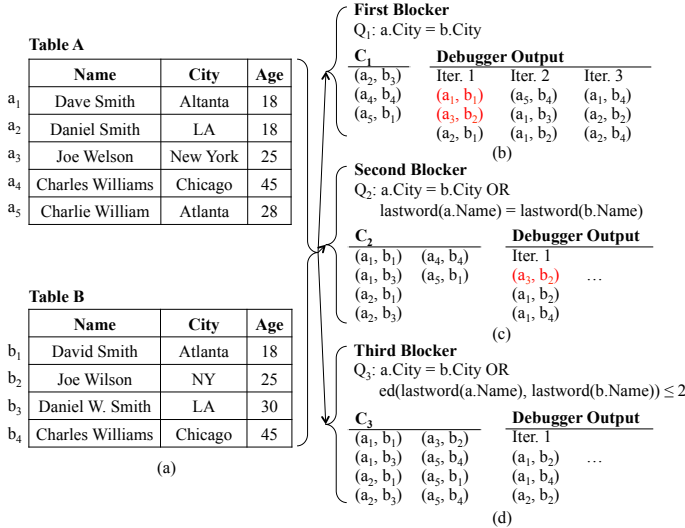


Figure 1: An example to illustrate MatchCatcher

U finds no true matches in this iteration, as well as in the third iteration.

At this point, U decides to stop looking for more killed-off matches, to focus on revising blocker Q_1 to improve its recall. U observes that the problem with pair (a_1, b_1) , which disagree on “City”, can be fixed by adding a new hash blocker that blocks on the last word of “Name”, i.e., keeps a tuple pair if they agree on this word (which is typically the last name). Figure 1.c shows Q_2 , the revised blocker, which is the union of two hash blockers.

Invoking MatchCatcher for Q_2 produces the list shown under “Debugger Output, Iter 1” in Figure 1.c. This list shows that while the new blocker Q_2 successfully keeps (a_1, b_1) , it still kills off (a_3, b_2) , a true match. A closer examination reveals that this is due to a misspelt last word: “Welson” vs. “Wilson”.

To fix such misspelling problems, U decides to keep a tuple pair if the last words of “Name” are very similar, e.g., within an edit distance of 2. This produces blocker Q_3 in Figure 1.d. Here, the hash blocker $lastword(a.Name) = lastword(b.Name)$ has been replaced by $ed(lastword(a.Name), lastword(b.Name)) \leq 2$, a more general blocker where “ed” computes the edit distance. Invoking MatchCatcher for Q_3 brings back no true matches, even after several iterations. Thus, user U stops, deciding to use Q_3 as the final blocker for A and B .

It is important to emphasize that MatchCatcher works with any of the current blocker types. Indeed, it requires as input only the two tables A and B and the set C resulting from applying the target blocker to the tables. MatchCatcher thus is *blocker independent*. We intentionally designed MatchCatcher this way to maximize its real-world applicability, i.e, to make sure that no matter which blocker a user has created, he or she can use MatchCatcher. Subsequent work will examine extending MatchCatcher to exploit the particularities of a specific blocker type.

Further, MatchCatcher does not estimate the *actual* recall, i.e., the fraction of matches surviving blocking. Doing so would require it to know the set of true matches in $A \times B$, which would be solving the EM problem itself! Indeed, MatchCatcher does not attempt to match A and B . Instead, its goal is to quickly find a large set of plausible matches killed off by the blocker and bring them to the user’s attention, so that the user can examine them to find true matches, get a sense about whether the blocker kills off too many such matches, and if so, what the problems are, so that

he/she can fix them. Section 6 shows that real-world users indeed find MatchCatcher very helpful in answering these questions.

Challenges: While promising, developing MatchCatcher raises difficult challenges. First, we must quickly search the vast space $D = A \times B - C$ (where C is the blocker’s output) to find plausible matches killed off by the blocker, and we must do so without materializing D . This search is further complicated by the fact that at this point MatchCatcher does not even know what it means to be a match (only the user knows). To address these problems, we observe that matching tuples tend to have similar values for certain attributes (e.g., Name, City). So we convert each tuple into a string that concatenates these attributes, e.g., converting tuple a_1 of Table A in Figure 1.a into “Dave Smith Atlanta”. We then perform a *top-k string similarity join (SSJ)* to find the k tuple pairs with the highest score with respect to these strings, and output these pairs as plausible matches. The state-of-the-art solution for top-k SSJs [34] proves too slow for our interactive setting. So we develop a new solution that is significantly faster.

Second, to find as many plausible matches as possible, we need to repeat the above procedure, but for different sets of attributes (e.g., find tuple pairs that are similar with respect to Name only, City only, both Name and City, etc.). We cannot consider all such sets, called *configs*, as there are too many. So we develop a solution to find a good set of configs.

Third, we must perform multiple related top-k SSJs, one for each config. This raises the challenge of how to perform them jointly across the configs. We develop an efficient solution that perform them in parallel on multiple cores yet reuse computations across the joins.

Finally, top-k SSJs over the configs produce a large set E of plausible matches (e.g., in the thousands). We cannot realistically expect the user to examine all of these matches. So we develop a solution that uses rank aggregation and active/online learning to rank the pairs in E , show the top n pairs to the user, ask him/her to identify the true matches, use this feedback to rerank the pairs, and so on, until the user has been satisfied or a stopping condition is reached. In summary, we make the following contributions:

- We show that debugging blocker accuracy is critical for EM.
- We describe MatchCatcher. As far as we know, this is the first in-depth solution to address the above problem. Our solution advances the state of the art in top-k string similarity joins, and exploits active/online learning to effectively engage with the user.
- Over the past two years, MatchCatcher has been successfully used by 300+ students in data science projects and by 7 teams at 6 organizations. We briefly report on this experience. We also describe extensive experiments showing that MatchCatcher is highly effective in helping users develop blockers, and that it can help improve the accuracy of even the best blockers manually created or automatically learned.

2 DEBUGGING BLOCKER ACCURACY

In this section we show that debugging blocker accuracy is critical for EM, discuss the limitations of current solutions, then provide an overview of the MatchCatcher solution.

Entity Matching (EM): This problem has received significant attention (see [6, 14, 30] for recent books and surveys). Many EM scenarios exist, e.g., matching two tables, matching within a table,

matching a table with a knowledge base, etc. [6]. In this paper, as a first step, we will consider the common EM scenario that matches two tables A and B , i.e., finds all tuple pairs $(a \in A, b \in B)$ that refer to the same real-world entity.

Types of Blockers: As discussed in the introduction, for large tables A and B we typically perform EM by creating a blocker Q , apply Q to A and B to produce a relatively small set of tuple pairs C , then apply a matcher to pairs in C . Over the past few decades blocking has received much attention. The focus has been on developing different blocker types and scaling up blockers, e.g., [18, 22, 33] (see [7, 13] for surveys).

Many blocker types have been developed. MatchCatcher works with all of them. In what follows we briefly discuss the most important types, as Section 6 experiments with many of them.

Well-known blocker types are attribute equivalence, hash, and sorted neighborhood. *Attribute equivalence (AE)* outputs a pair of tuples if they share the same values of a set of attributes (e.g., blocker $Q_1: a.City = b.City$ in Figure 1.b). *Hash blocking* (also called *key-based blocking*) is a generalization of AE, which outputs a pair of tuples if they share the same hash value, using a pre-specified hash function. For example, blocker Q_2 in Figure 1.c combines the hash blocker $lastword(a.Name) = lastword(b.Name)$ and the AE blocker Q_1 . *Sorted neighborhood* outputs a pair of tuples if their hash values (also called *key values*) are within a pre-defined distance.

More complex types of blockers include similarity- and rule-based [6, 8, 18]. *Similarity-based blocking (SIM)* is similar to AE, except that it accounts for dirty values, misspellings, abbreviations, and natural variations by using a predicate involving string similarity measures, such as edit distance, Jaccard, overlap, etc. [36]. Examples include $ed(lastword(a.Name), lastword(b.Name)) \leq 2$, a blocker which outputs tuple pairs where the last words of their names have an edit distance of at most 2, and blocker $jaccard(a.title, b.title) \geq 0.4$, which outputs pairs of books whose titles have a Jaccard similarity score of at least 0.4. *Rule-based blocking* is perhaps most general. It outputs a tuple pair satisfying a rule or a set of rules encoding domain heuristics, e.g., blocker Q_3 in Figure 1.d consists of two rules. Such blockers can be viewed as the union of multiple blockers, one per rule.

Other types of blockers include phonetic (e.g., soundex), suffix-array, canopy, etc. (see [6, 14] for an extensive discussion).

Efficient Execution of Blockers: Efficient techniques have been developed to execute the above blocker types, both on a single machine and a cluster of machines (e.g., [8, 18, 22]). To execute hash/AE blocking, we partition the tuples in A and B into *blocks*, such that all tuples in each block share the same hash value, then output only pairs of tuples that are in the same block.

To execute a SIM blocker, e.g., $ed(lastword(a.Name), lastword(b.Name)) \leq 2$, we build an index I (e.g., prefix filtering index [36]) on the tuples in A , say. Next, for each tuple $b \in B$, we consult I to identify all tuples $a \in A$ such that the pair (a, b) can possibly satisfy $ed(lastword(a.Name), lastword(b.Name)) \leq 2$. We check if (a, b) indeed satisfies this predicate, and if yes, then output the pair. Many efficient string indexing techniques [36] can be used to implement SIM blockers. Recent work [8] has also discussed efficient techniques (e.g., using indexing and MapReduce) to execute rule-based blockers.

Accuracy of Blockers: Blocker accuracy is typically measured using recall, defined as follows:

Definition 2.1. [Blocker recall] Suppose applying blocker Q to two tables A and B produces the output C . Let $M \subseteq A \times B$ be the (unknown) set of true matches between A and B , then $recall(Q) = |M \cap C|/|M|$.

Due to dirty data, misspellings, natural variations, synonyms, missing values, etc., no single blocker type produces the highest recall on all datasets. In fact, on any particular dataset, blockers can vary drastically in recalls (e.g., 2.5-98.2% in our experiments).

Finding a blocker with high recall, however, is critical for many EM applications. Counter-terrorism EM applications often need very high coverage, i.e., finding *all* person descriptions that match, and thus want 100% blocking recall. Similar high-coverage examples arise in fraud detection, e-commerce, law, medicine, insurance, and pharmaceutical industry, among others. EM applications with inherently small numbers of matches naturally do not want the blocker to kill off many of these. Finally, EM applications often compute statistics over the matches (e.g., the percentage of patients attending both hospitals), which can be seriously distorted by blockers with low recall.

Limitations of State of the Art: As a result, the topic of blocker accuracy has received growing attention. Proposed solutions include combining multiple blockers to maximize recall (e.g., [12, 20, 22]), and using a sample of tuple pairs labeled as match/no-match to learn blockers with high recall [2, 8, 18, 25].

While promising, these solutions can still produce blockers with varying recalls, oftentimes falling short of 100%. For example, a recent work [8] shows that extensive manual effort to combine hash blockers achieves only 38.8% and 72.6% recall on two datasets. (Obviously we cannot combine *all* possible blockers as there are too many of them.) Another recent work [18] learns blockers using samples labeled by crowdsourcing, but achieves only 92% recall on a data set. In general, due to the difficulties in obtaining a good sample, sampling flukes, etc., today there is still no guarantee that a blocker learned on a *sample* provably achieves high recall when applied to the original tables.

Since there is still no “fool-proof” method to develop a blocker with high recall, it follows that given a blocker Q (either created manually or learned), it is still highly desirable to know how well Q does recall-wise, and what the possible problems are, so that we can improve it. MatchCatcher helps answer these questions, and thus can be considered *complementary* to the above solutions. For example, Section 6 describes a scenario where after the solution in [8] had been used to learn a blocker, we applied MatchCatcher to this blocker and uncovered multiple problems, which can be addressed to further improve the blocker recall.

Overview of MatchCatcher: As discussed, MatchCatcher addresses the following problem:

Definition 2.2. [Finding killed-off matches] Let C be the output of applying blocker Q to tables A and B . Then $D = A \times B - C$ is the set of all pairs killed off by Q . Help the user quickly find as many true matches as possible in D (without materializing it). Examining these matches helps the user understand how well Q does recall-wise, and what can be done to improve its recall.

Figure 2 shows the architecture of MatchCatcher. Given two tables A and B , the Config Generator examines the two tables to generate a set of configs, each of which is a set of attributes (e.g., $\{Name, City\}$). For each config g , the Top-k SSJs module performs a top-k string similarity join to find the k tuple pairs that (a) have the highest score with respect to the attributes in g , and (b) are killed off by blocker Q . Note that to check Condition

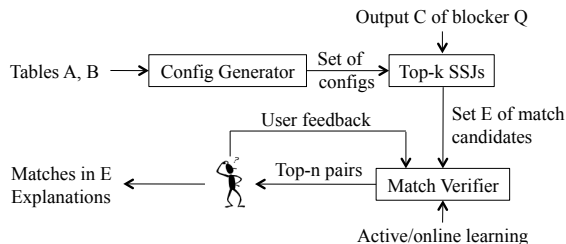


Figure 2: The MatchCatcher architecture

(b), this module does not need to know Q . It only needs to know C , the output of applying Q to A and B . Hence MatchCatcher works independently of the blocker type.

The Top- k SSJs module sends all top- k lists (one per config) to the Match Verifier. This module uses a rank aggregator to combine the lists into a single global list, shows the top n pairs to user U , asks U to identify true matches, uses this feedback together with active and online learning to rerank the pairs, and so on, until U is satisfied or a stopping condition is met. U can examine the true matches to understand how well blocker Q does recall-wise, and to obtain explanations for why these matches are killed off. This helps U decide if Q should be revised, and if so, then how. The next few sections describe MatchCatcher in detail.

3 GENERATION OF CONFIGURATIONS

We now describe the Config Generator, which outputs a set of configs, each being a set of attributes. We cannot consider all possible configs, so the key challenge is to select a good subset of configs. We show how to do so, by carefully managing attributes with many missing values, few unique values, or long string values.

3.1 How Configurations Are Used

We first motivate the notion of configurations (or “configs” for short) and explain how they are being used. Later we build on this to discuss how to select a good set of configs.

Recall that we want to quickly search $D = A \times B - C$, the set of tuple pairs killed off by blocker Q , to find pairs that can be matches. This raises three problems. First, D is not materialized, we only have A , B , and C . Second, even if D is materialized, it would be too large to search quickly. Finally, we do not even know what to search for, since at this point MatchCatcher does not know what a match is (only the user knows).

To address these problems, we begin by assuming that tables A and B share the same schema S (extending MatchCatcher to the case of different schemas is future work). We observe that matching tuples tend to share similar values in a set of attributes, say g (e.g., $\{Name, City\}$). So we want to quickly find tuple pairs in D that share similar values for g and return those as possible matches.

To do so, we convert each tuple a in A into a string $str_g(a)$ that concatenates the values of all attributes in g . For example, if a is (David Smith, Atlanta, 43) and $g = \{Name, City\}$, then $str_g(a)$ is “David Smith Atlanta”. This converts Table A into a set A_g of such strings. We convert Table B into a set B_g of strings similarly.

Let $h(x, y)$ be a string similarity measure which computes a score in $[0, 1]$ between two strings x and y . Examples of such measures are Jaccard, cosine, overlap, edit distance, etc. [36]. Then next we perform a top- k string similarity join (SSJ) between A_g and B_g to find the k tuple pairs in $A \times B$ with the highest $h(x, y)$ score. Techniques have been developed to quickly perform top- k

SSJs [34, 37]. Of course, our goal is not to find pairs in $A \times B$, but rather in $D = A \times B - C$. We can modify the above techniques slightly to ensure this, by dropping a found pair if it is in C . We then return the k pairs in D with the highest $h(x, y)$ score as possible matches.

The above procedure does not require a materialized D , only tables A , B , and C (the output of blocker Q). It can quickly search D using a modified version of top- k SSJs to return possible matches. Of course, at this point we still do not know if these are indeed matches. But later we can work with the Match Verifier to quickly shift through them to find true matches, if any. We now discuss several important aspects of the above procedure.

Why Concatenating the Attributes? We can use a variety of methods to find tuples that share similar values for attributes in g , e.g., finding pairs that share similar values for *each* attribute in g , then taking their intersection, say. However, given the interactive nature of debugging, we want this step to be as fast as possible. Hence we decide not to treat the attributes in g separately, but concatenate all of them into a single string, then compare them using SSJs. Section 4 shows that this method can quickly search a very large set D . But a drawback is that we can return *false positives* such as tuple pair (Jim Madison, Smithville, 32) and (Jim Smith, Madison, 32), because their concatenated strings are very similar given certain similarity measures. Such false positives, however, can be “weeded out” in the Match Verifier, using user feedback and active/online learning (see Section 5).

Which String Similarity Measure to Use? Given that similar attribute values can still vary significantly (e.g., “Dave Smith” vs “David Frederic Smith”), measures that treat strings as sets (e.g., Jaccard, cosine, etc.) typically work better than those that treat strings as sequences of characters (e.g., edit distance) [9]. So for MatchCatcher, we use the well-known Jaccard measure that tokenizes two strings x and y into two sets of words P_x and P_y , then returns $|P_x \cap P_y| / |P_x \cup P_y|$ [34]. However, Theorem 4.2 shows that our solution can also work with other set-based similarity measures, namely overlap, cosine, and Dice [34].

Why Multiple Configurations? So far we have used just one config g to find match candidates. Using multiple configs, however, can produce more matches. For example, config $\{Name, City\}$ may not return the pair (David Smith, Seattle) and (Dave Smith, Redmond) because the cities are different. Config $\{Name\}$ however can. Conversely, config $\{Name\}$ may not return the pair (Chuck Smith, San Francisco) and (Charles F. Smith, San Francisco) because the names are too different, but config $\{Name, City\}$ can. Together, these two configs can return more matches than either of them in isolation. Generating a good set of configs however is a major challenge, which we address next.

3.2 Generating Multiple Configurations

As a baseline, we can use all subsets of attributes in S (the schema of A and B) as configs. But this generates too many configs even for a moderate size (e.g., $|S| = 8$ produces $2^{|S|} - 1 = 255$ configs). We cannot use all of them because the total SSJ time would be too high. So we must select a smaller set of configs.

To do so, we select a set of promising attributes in S , then use them to generate configs, in a top-down fashion. In each step of the process, we select which configs to generate next by carefully considering the impact of attributes with many missing values, few unique values, or long string values. The end result is a *config tree* consisting of multiple configs. Later the Top- k SSJs module

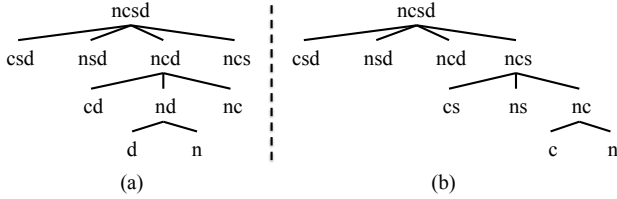


Figure 3: An example of generating config trees.

will traverse this tree to perform top-k SSJs on the configs in a joint fashion. We now elaborate on these steps.

Selecting the Most Promising Attributes: We first classify attributes in S as string, numeric, categorical, and boolean, using a rule-based classifier. Next, we drop numeric attributes (e.g., Salary, Price) because matching tuples still often differ in their values (e.g., the same product having different prices). Finally, we drop categorical and boolean attributes whose appearances in tables A and B are different. For example, if Gender has values $\{Male, Female\}$ in A but $\{M, F, U\}$ in B , then we drop it as these two sets share no value (in general if the Jaccard score of these two sets is less than a pre-specified threshold then we drop the attribute). The remaining attributes are string, or categorical/boolean but with similar sets of values. We return these as T , the set of the most promising attributes to be used for config generation. (Of course, the user can also manually curate schema S to generate T . The experiments in Section 6 however do not involve any manual curation.)

Generating a Config Tree: Given the set T of promising attributes, we generate a *config tree* in a top-down fashion, then return all configs in the tree. Specifically, we start with T as the config at the root of the tree. Next, we “expand” this node by removing each attribute from T to obtain a smaller config of size $|T| - 1$. This produces $|T|$ new configs, which form the nodes at the *next level* of the tree. We then select just one node at this level to “expand” further, and so on (we will discuss how to select shortly). This continues until we have reached configs of just one node. Figure 3.a shows an example config tree, assuming $T = \{n, c, s, d\}$ (which stand for Name, City, State, and Description, respectively).

Intuitively, this strategy ensures that we generate a diverse set of $|T|(|T| + 1)/2$ configs of varying size $|T|, |T| - 1, \dots, 1$. The config tree will also be used to guide the joint execution of top-k SSJs on the configs (see Section 4.2). We now turn to the challenge of how to select a node to expand in the config tree.

Managing Many Missing Values and Few Unique Values: Consider again the config tree in Figure 3.a. Suppose we are currently at the second level of the tree, and need to select one node among the four nodes csd , nsd , ncd , and ncs , to expand. This selection is equivalent to *selecting an attribute to exclude from subsequent config generation*. Indeed, if we exclude attribute s , then we select node ncd to expand (as shown in the figure). Otherwise if we exclude d , then we select the rightmost node ncs to expand, and so on.

So which attribute should we exclude? We observe that if an attribute has many missing values, then keeping it for subsequent config generation is not desirable, because we will end up with configs that produce similar top-k lists. For example, suppose we have selected config ncd to expand (as shown in Figure 3.a), and suppose that d has many missing values, then many strings for config ncd and config nc will be identical, potentially leading to similar top-k lists. In the extreme case, if all values for d are

Name: Bryan Lee, **City:** Austin, **State:** TX,
Desc: Joined in 8/2003, promoted to team lead 5/2005, promoted to director of sales 4/2009. Currently on unpaid leave until 1/2013.

Name: Bryan M. Lee, **City:** Austin, **State:** TX,
Desc: Outstanding customer service record 03-05. Achieved sales of \$2M/year 05-09. Shortlisted for VP of sales 2011. Shortlisted for VP of marketing 2012.

Figure 4: Examples of tuples with long string attributes.

missing, then these two top-k lists are identical. Clearly, we want different configs to produce substantially different top-k lists, to avoid redundant work and to maximize the number of matches we can retrieve.

Another observation is that if an attribute has more unique values than another, e.g., c vs s (which stand for City and State, respectively), then it is better to exclude s , the one with fewer unique values, because intuitively, if two tuples agree on c , they are more likely to match than if they agree on s , all else being equal. Thus, to maximize the number of matches we can retrieve, we should strive to keep the “more specific” attributes, i.e., the ones with more unique values.

Combining the above two observations, we define the *e-score* (shorthand for “expected benefit”) of an attribute as follows:

*Definition 3.1. [E-score of an attribute] Let $n_A(f)$ be the ratio of the number of non-missing values of attribute f in A over the number of tuples in A , and $u_A(f)$ be the ratio of number of unique values of f in A over the number of non-missing values of f in A . We define $n_B(f)$ and $u_B(f)$ similarly. Define $e_A(f) = 2n_A(f)u_A(f)/[n_A(f) + u_A(f)]$ and define $e_B(f)$ similarly. Then we define the *e-score* of attribute f as $e(f) = e_A(f)e_B(f)$.*

We then select the attribute with the lowest *e-score* to exclude at each level of the config tree. For example, suppose $e(n) > e(d) > e(c) > e(s)$. Then at the second level of the tree in Figure 3.a, we exclude attribute s , which means selecting node ncd to expand. At the third level of the tree, we exclude c , which means selecting node nd to expand.

Managing Long String Attributes: Many datasets contain attributes with long string values, e.g., Comment, Desc, etc. Figure 4 shows two tuples where attribute Desc has such long values. Such long attributes can cause two problems. First, they can cause multiple configs to generate very similar top-k lists.

Example 3.2. Consider again the config tree in Figure 3.a. Suppose attribute d has long string values (such as those shown in Figure 4). Then all seven configs involving d can generate similar top-k lists because the long values of d “overwhelm” the short values of the remaining attributes. So when moving from a config involving d to another (e.g., from ncd to nd), the strings do not change much, and therefore their similarity scores also do not change much (we formalize this notion below), leading to similar top-k lists.

The second problem is that if the long string values are different for matching tuples, then a config involving this long attribute will fail to return the match. For example, the two tuples in Figure 4 match, but any config involving attribute Desc will not return this match, because the values for Desc here are very different, and so the score between the two tuples will be low.

To address this, we modify our config-tree generation procedure as follows. Suppose we need to select a config node in the tree to expand. Before, we select $g_{default}$, the one without the attribute with the smallest *e-score*. Now, we first run a procedure FindLongAttr to see if there is any attribute that is “too long” (i.e., likely to adversely affect selecting good configs). If such an

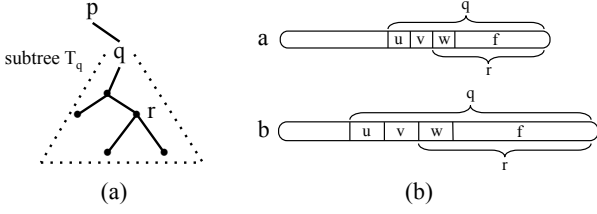


Figure 5: Finding attributes judged too long.

attribute f_{long} exists, then we select the config without f_{long} to expand. Otherwise we select $g_{default}$, as usual.

Example 3.3. Consider again Figure 3.a, which shows the “default” config tree with root $ncsd$. To handle long attributes, once we are at the second level, we do not automatically select ncd (the config without s , the attribute with the smallest e -score) for expansion. Instead, we run FindLongAttr at this level. Suppose it returns d (thus judging d to be too long). Then we select ncs , the config without d , for expansion. This produces new configs cs , ns , and nc (see Figure 3.b). Suppose running FindLongAttr at the level of these new configs returns no attribute. Then we select config nc (the config without s , the attribute with the smallest e -score) for expansion (see Figure 3.b).

We now explain procedure FindLongAttr. The key challenge is to formalize what it means to be “too long”. Let p be a node in the config tree. Suppose that when running the default config generation procedure (the one that does not consider long attributes), we end up selecting q , a child node of p , for expansion, and that we eventually generate a subtree T_q rooted at q (see Figure 5.a).

We say that an attribute f is too long if it “overwhelms” many config nodes in subtree T_q , specifically if it overwhelms at least half of the configs in $F(T_q)$, the set of configs in T_q that contain f . In turn, we say that f overwhelms a config $r \in F(T_q)$ (see Figure 5.a) if the top- k list obtained from config r is “roughly the same” as the top- k list obtained from config q (we formalize this below). Intuitively, we want to avoid such cases, because we want each config to return a different top- k list, to maximize the number of true matches that we will find. So if we find that f overwhelms at least half of the configs in $F(T_q)$, then we judge f to be too long and should be removed. That is, instead of selecting q for expansion, we will select the config (in the same tree level as q) that does not contain f .

Of course, we do not have access to the top- k lists of r and q . So we develop a condition which if true would suggest that the two lists are “roughly the same”. Specifically, let $sim_g(a, b) = h(str_g(a), str_g(b))$ be the string similarity function between the string values of two tuples a and b , for config g . Suppose that for all tuple pairs (a, b) in $D = A \times B - C$ we have

$$\text{Condition 1: } |sim_q(a, b) - sim_r(a, b)| / sim_q(a, b) \leq \alpha,$$

for a small pre-specified α value, say 0.2. Then we can say that when we switch config from q to r , the score of each tuple pair does not change much, so the top- k list for r will stay roughly the same as that of q .

Checking Condition 1 for all pairs (a, b) in D is not feasible. Hence we perform a theoretical analysis for an idealized scenario (described below). Of course, this idealized scenario rarely happens in practice. But understanding it helps us come up with an efficient heuristic to check Condition 1.

Let $L_f(a)$ be the length (i.e., the total number of words) of attribute f in tuple a , $L_q(a)$ be the sum of the lengths of all attributes in q , for tuple a , and so on. The idealized scenario

assumes that (a) attribute f takes the same proportion of the total length of q in both a and b , i.e., $L_f(a)/L_q(a) = L_f(b)/L_q(b) = \beta$, and (b) the remaining length of q is equally distributed among the remaining attributes of q , i.e., $L_k(a) = [(1-\beta)L_q(a)]/(|q|-1)$ for all attribute k in $q - \{f\}$, and the same condition applies to tuple b .

Example 3.4. Consider the two tuples a and b in Figure 5.b, where $q = \{u, v, w, f\}$ and $r = \{w, f\}$. We assume that $L_f(a)/L_q(a) = L_f(b)/L_q(b) = \beta$, and $L_u(a)/L_q(a) = L_v(a)/L_q(a)$ and $L_u(b)/L_q(b) = L_v(b)/L_q(b)$.

Then we can show that (see [24] for a proof sketch):

THEOREM 3.5. Let $a \in A$ and $b \in B$ be two tuples that satisfy the above assumptions. If

- (R_1) $sim_q(a, b) \geq [\sqrt{(1+\alpha)^2 + 8} - (1+\alpha)]/4$, and
- (R_2) $\beta \geq 1 - \frac{(|q|-1)}{|q \setminus r|} \cdot \frac{\alpha}{(1+\alpha)} \cdot \frac{\max\{L_q(a), L_q(b)\}}{L_q(a)+L_q(b)}$,

then pair (a, b) satisfies Condition 1.

Intuitively, this theorem says that if $sim_q(a, b)$ is sufficiently high (Requirement R_1), and attribute f is sufficiently long (Requirement R_2), then $sim_r(a, b)$ will be close to $sim_q(a, b)$. It is not difficult to show that the quantity on the right-hand side of R_1 is upper bounded by 0.5. In practice, we observe that users typically examine only the top few tens of pairs in each top- k list (see Section 5), and that if these pairs are true matches, their scores often exceed 0.5, making R_1 true. As a result, if R_2 is also true, then attribute f is long enough to “overwhelm” these pairs. That is, these pairs will change little score-wise when switching from config q to r , thus typically will still show up in the top few tens of pairs of the top- k list for r , an undesirable situation.

To avoid such situations, we will focus on checking R_2 . Checking R_2 for many pairs (a, b) is not practical. So we approximate this checking using average lengths, i.e., we (a) replace β in the left-hand side of R_2 with $\min\{AL_f(A)/AL_q(A), AL_f(B)/AL_q(B)\}$, where $AL_f(A)$ for example is the average length of attribute f in Table A, and (b) replace $L_q(a)$ and $L_q(b)$ in the right-hand side of R_2 with $AL_q(A)$ and $AL_q(B)$, respectively.

Procedure FindLongAttr then works as follows. Suppose we have selected config q for expansion (because it does not contain s , the attribute with the least e -score). Then for each attribute f (other than s), we (a) identify $F(T_q)$, the set of configs in T_q that contain f , (b) declare f “too long” if the above approximate checking is true for at least half of the configs $r \in F(T_q)$. It is not difficult to prove that at most one attribute f will be found too long. If so, we do not select q , but select instead the config that does not contain f for expansion. Otherwise, we select q , as usual. This procedure takes less than a second in our experiments.

Discussion: Note that we do not completely remove attributes with many missing values, few unique values, or long values from config generation. Instead, each such attribute f may be removed only at some point during the generation process. Configs generated earlier still contain f .

Further, our work here is related to, but very different from work such as [3, 10]. Those works find attributes that are discriminative for classification, often using a labeled sample (as many works in learning do). Here we do not look for discriminative attributes. Instead, we look for attributes such that if two tuples agree on their values, then they are likely to match. For example, suppose all tuples in table A have the same value “US” for “Country”, and all tuples in table B have the same value “Canada”.

Then “Country” is a discriminative attribute because if two tuples disagree on it, they definitely do not match. For our purpose, however, “Country” has little expected benefits, because if two tuples agree on it, it is still not likely that they match (not as much as if they agree on “State” and “City” say).

In fact, the work [29] also treats attributes with missing values and few unique values in a way similar to ours (for blocking and matching). However, it does not handle long attributes, and uses only one config, and thus is significantly outperformed by MatchCatcher (see Section 6).

4 TOP-K STRING SIMILARITY JOINS

So far we have discussed generating a set of configs. We now discuss performing top-k SSJs over these configs (one per config). Previous work has discussed top-k SSJs for a single config [34]. Here we significantly improve that work (and our solution can be applied to top-k SSJ situations beyond this paper). We then discuss executing multiple top-k SSJs jointly, by reusing results across the configs, in a parallel fashion.

MatchCatcher currently works with the Jaccard string similarity measure, and we will explain it using that measure. However, it is important to note that all algorithms discussed below also work with the set-based similarity measures cosine, overlap, and Dice.

4.1 Improving Top-k Join for a Single Config

As far as we can tell, the state of the art in top-k SSJs is TopKJoin [34]. Given a set J of strings, TopKJoin finds the k string pairs with the highest similarity scores, for a pre-specified k , in a branch-and-bound fashion. Specifically, it maintains a prefix for each string in J , incrementally extends these prefixes, finds string pairs whose prefixes overlap, computes their similarity scores, use these scores to maintain a top-k list, then extends the prefixes again, and so on.

Example 4.1. Suppose J consists of the four strings w, x, y, z in Figure 6.a. We begin by creating a prefix $p(w) = “a”$ for w , then a prefix $p(x) = “a”$ for x . At this point the prefixes of the pair (x, w) overlap. Hence we compute the Jaccard score 0.8 for this pair, then initializes the top-k list K to be containing just this pair. (Here we assume $k = 2$.)

Next, we create prefix $p(y) = “b”$. This does not produce any new pair whose prefixes overlap. So we continue by creating prefix $p(z) = “b”$. This produces a new pair whose prefixes overlap: (z, y) with score 0.43. Figure 6.b shows the updated top-k list K .

Next, we select one prefix to extend (we will discuss shortly how). Suppose we select $p(x)$ and extend it by one token. Then $p(x) = “ab”$ (see Figure 6.a). This produces two new pairs whose prefixes overlap: (x, y) with score 0.67 and (x, z) with score 0.43. Figure 6.c shows the updated top-k list K . We then select another prefix to extend, and so on. Finding new pairs with overlapping prefix can be done efficiently using an inverted index from token to the prefixes of the strings [34].

We now discuss how to select a prefix to extend. Suppose we have imposed a global ordering on all tokens, and sorted the tokens in each string w, x, y, z in that order (see Figure 6.a). Suppose also that we have created prefixes of size 1, namely $p(w) = “a”, p(x) = “a”, p(y) = “b”, p(z) = “b”$, and are now deciding which prefix to extend. Suppose we select $p(w)$ and extend it by one token, to be “ab”. Then it is easy to show that the scores of all new pairs generated by this extension are capped by 0.75. Indeed, any new pair must involve w . Let such a pair

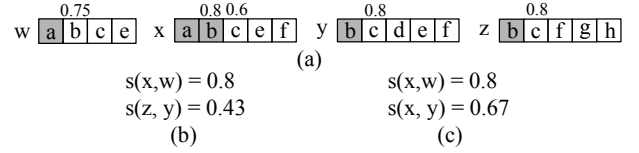


Figure 6: An illustration of top-k computation.

be (w, v) . Then the first common token that they share should be “b” (the token just being added to $p(w)$). So they can share at most this token b and the remaining “unseen” tokens of w . Thus $|w \cap v| \leq 3$. Since $|w \cup v| \leq |w| = 4$, it follows that the Jaccard score of (w, v) is capped by $3/4=0.75$. We write 0.75 on top of token “b” in w to indicate that when we extend $p(w)$ to include this token, the score of any new pair generated by TopKJoin will be capped by 0.75. Similarly, we can write 0.8 for the second tokens of x, y, z (see Figure 6.a).

We then select the prefix that when extended will include the token with the highest “cap” number (in the hope that it will generate new pairs with the highest possible scores). In this case, we select $p(x)$ (but $p(y)$ and $p(z)$ also work).

We now discuss how to stop. Observe that the “cap” number for “c” in x is 0.6. By the time we have to consider whether to extend $p(x)$ to include “c”, the top-k list already has a lower-bound score of 0.67 (see Figure 6.c), greater than 0.6. As a result, we do not have to extend $p(x)$ to include “c”, and in fact, prefix extension on x can be stopped at this point. TopKJoin terminates when all prefix extensions have stopped, either early (as described above) or because the prefix has covered the entire string. The paper [34] describes TopKJoin in detail, including optimizations to avoid redundant computations.

The QJoin Algorithm: TopKJoin has a major limitation. Every time it generates a new pair (u, v) , it immediately computes the similarity score of (u, v) (then updates the top-k list). Computing this score turns out to be very expensive, especially if u and v are long strings. In a sense, it is also “premature”, because it can be shown that when we generate (u, v) (as a new pair), we only know that they share a single token. There is no evidence yet that they share more tokens and thus are likely to have high similarity score. If they indeed share only one or few tokens, and yet we still compute their score, then that score is likely to be low. So the pair will not make it into the top-k list, yet we have wasted time computing its score.

To address this problem, when generating new pairs, we do not immediately compute their scores. Instead, we keep track of the number of common tokens each pair has, and update this number whenever a prefix is extended. We then compute the score of a pair only if it has q common tokens, and thus is likely to have a high score. It is difficult to select q analytically, so we select it empirically as follows. Assuming at least four CPU cores, we begin by running the modified TopKJoin for $q = 1, q = 2$, etc., on all cores, one q value for each core, for $k = 50$. (Note that TopKJoin always does $q = 1$.) Then whichever core finishes first, we keep the process on that core running to produce the rest of the top-k list (effectively selecting the q value associated with that core), and kill the processes on the other cores.

It is straightforward to adapt the above algorithm to work with two tables (instead of just one), and to remove a pair from the top-k list (during the top-k computation) if it happens to be in the candidate set C . Henceforth we refer to this new algorithm as QJoin.

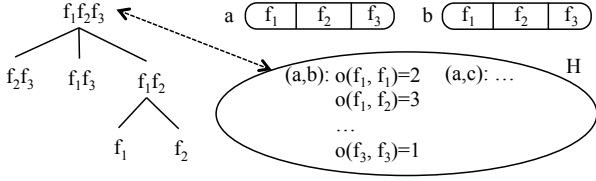


Figure 7: Reusing across top-k computations.

4.2 Joint Top-k Joins Across All Configs

TopKJoin can only be applied to a single config [34]. Our setting however involves multiple related configs. We now describe a solution to find top-k lists *jointly across the configs*. To do so, we use QJoin, but modify it to reuse similarity score computations and top-k lists across the configs, and process the configs in parallel.

Reusing Similarity Score Computations: As discussed in Section 4.1, computing the similarity score of a pair (a, b) is very expensive, especially for long strings. Hence, we want to reuse such computations across the configs. To do so, we process the configs in the config tree in a breadth-first order, e.g., processing the root config $f_1f_2f_3$ of the config tree in Figure 7 (where the f_i -s are attributes), then the next-level configs, f_2f_3, f_1f_3, f_1f_2 , and so on.

When processing a config g (i.e., finding its top-k list), we keep track of certain information, then reuse it when processing configs in the subtree of g . For example, consider again the config tree in Figure 7. We start by tokenizing the strings wrt the root config $f_1f_2f_3$ into multisets of word-level tokens. Next, we process the config $f_1f_2f_3$. This process computes the Jaccard score of multiple tuple pairs. When computing the score of such a pair, say (a, b) , we compute and store the number of overlapping tokens between any two attributes f_i of a and f_j of b in an in-memory database H . Figure 7 illustrates this step. Here, $o(f_1, f_1) = 2$ means attributes f_1 of a and f_1 of b share two tokens. (We only store in H attribute pairs that share tokens.)

Then we can reuse H to drastically speed up processing configs in the subtree rooted at $f_1f_2f_3$. For example, consider processing config f_1f_2 . If during this process we need to re-compute the score of (a, b) (now with respect to only f_1 and f_2), then we can use H to compute $Overlap_{f_1f_2}(a, b) = o(f_1, f_1) + o(f_1, f_2) + o(f_2, f_1) + o(f_2, f_2)$, then compute the above score as

$$Overlap_{f_1f_2}(a, b) / (L_{f_1f_2}(a) + L_{f_1f_2}(b) - Overlap_{f_1f_2}(a, b)),$$

where $L_{f_1f_2}(a)$ for instance is the length in tokens of the concatenation of f_1 and f_2 for a . Computing the score of (a, b) this way is far faster than computing from scratch.

Note that while processing config f_1f_2 , if we have to compute the score of a new pair (c, d) not yet in H , then we will store similar overlap information for (c, d) in H , to enable reuse when processing configs in the subtree rooted at f_1f_2 , and so on.

Reusing Top-k Lists: When applying to a config g , algorithm QJoin starts with an empty top-k list K , then gradually grows K as it iteratively expands the prefixes. In our setting, however, since we process multiple configs, a promising idea is to use the top-k list of a previous config to initialize the top-k list of the current config.

For example, after processing config $g = f_1f_2f_3$ (Figure 7), we store its top-k list K_g . Then when processing config $h = f_1f_2$, we use the database H described earlier (which stores overlap information) to re-adjust all scores in K_g . This is necessary because these scores are computed wrt $f_1f_2f_3$, but now we want

them to be adjusted to consider only f_1f_2 . This re-adjustment is fairly straightforward (and inexpensive) because the overlap information for all pairs in K_g should already be in H . Next, we run the algorithm QJoin as usual to process config $h = f_1f_2$, but using the K_g list with the adjusted scores as the initial top-k list K_h (instead of using an empty list).

Observe that the above procedure enables reusing top-k lists from a parent to a direct child (e.g., from $f_1f_2f_3$ to f_1f_2). Reusing across the “siblings” appears much more difficult. For example, given the top-k list for f_1f_3 , there is no obvious way to quickly adjust its scores for f_1f_2 , using database H . Hence, currently we do not yet consider such sibling reuse.

Finally, reuse does not come for free. It helps avoid computing certain similarity scores from scratch, but incurs an overhead of storing and looking up the overlap information. If the tuples are short, then the overhead can easily overwhelm the savings. As a result, we trigger reuse only if the average tuple length is at least t tokens (currently set to 20).

Parallel Processing of the Configs: Finally we explore parallel processing on multiple cores. (We consider multicore single machines for now because it is a common setting for many domain science users [19].) An obvious idea is to process each config across multiple cores. For example, we can split Table A into two halves A_1 and A_2 and Table B into B_1 and B_2 , find the top-k list for A_1 and B_1 on the first core, the top-k list for A_1 and B_2 on the second core, etc., then merge the top-k lists. In practice, this approach suffers from severe skew: one core finishes quickly while another runs forever. While it is possible to split the tables intelligently to mitigate skew, this adds considerable overhead and implementation complexity.

As a result, we opted for processing one config per core. Specifically, we traverse the config tree breadth-first, and assign configs to cores in that order (when a core finishes, it gets the next config “in queue”). This solution *continuously utilizes all cores*. But it raises two problems. First, two configs (e.g., $f_1f_2f_3$ and f_1f_2) may concurrently write, or one reads and the other writes, into database H , causing concurrency control issues. To address concurrent writes, observe that only configs with non-empty subtrees (e.g., $f_1f_2f_3$ and f_1f_2 in Figure 7) will write. For each such config g , we require it to write into a separate in-memory database H_g .

To address dirty reads (e.g., $f_1f_2f_3$ writes into a database while f_2f_3 reads from it), we note that here each “write” just *inserts* a value; it never modifies or deletes. For such cases there are atomic hashmaps that perform *atomic inserts*, thus avoiding dirty reads. So we implement each database H_g as one such hashmap (using the Atomic Unordered Hashmap in Facebook’s C++ Folly package).

Finally, if a parent config, e.g., $g = f_1f_2f_3$, has not yet finished, then a direct-child config, $h = f_1f_2$, cannot reuse g ’s top-k list. In such situations, we start config h with an initial empty top-k list. When config g finishes, it sends its top-k list to h . Config h merges its current top-k list with the new top-k list from g , to obtain a potentially better top-k list, then continues. The technical report [24] shows the pseudo code of the complete algorithm, and the following theorem shows its correctness (see [24] for a proof sketch):

THEOREM 4.2. *Given two tables A and B , the output C of a blocker on A and B , a set of configs G , a string similarity measure which is Jaccard, cosine, overlap, or Dice, and a value k , the above algorithm returns a set of top-k lists, where each top-k list is the*

output of applying Algorithm QJoin to A, B , and C , using a config $g \in G$ and the given similarity measure and k value.

5 INTERACTIVE VERIFICATION

So far we have discussed processing configs to obtain a set of tuple pairs. We now discuss identifying true matches in this set, via user engagement, rank aggregation, and active/online learning.

Engaging the User: Let E be the union of the top- k lists obtained from processing all configs. Typically E is large (e.g., 3,011-7,089 in our experiments) and the true matches make up just a small portion of E . Thus expecting a user U to be able to examine the entire set E to find true matches is unrealistic.

A reasonable solution is to rank the pairs in E such that the true matches “bubble” to the top, then present the ranked list to U . However, our experiments with a variety of ranking methods (see below) suggest it is very difficult to do so. Typically, the top of the ranked list indeed contains multiple matches. But then the remaining matches tend to be scattered far and wide in the list.

As a result, we decided to engage user U : we rank the pairs in E , present the top- n pairs to U (currently $n = 20$), ask U to identify the true matches, use this feedback to rerank the list, then present the next top- n pairs to U , and so on. As such, we help U iteratively identify true matches, but use this identification to help “bubble” the remaining matches to the top of the ranking.

Using Rank Aggregation: Let m be the number of configs and L_1, \dots, L_m be the top- k lists obtained from these configs. To engage user U , we first need to aggregate these lists into a single list. Many aggregation methods exist, e.g., [4, 15]. Here we use MedRank [15], a popular method. To use MedRank, we first sort each list L_i in decreasing order of score, then associate each item in the list with a rank, i.e., an integer, such that the higher the score, the lower the rank and items with the same score receive the same rank. Next, we compute for each item a global rank which is the median of its ranks in the lists. Finally, we sort the items in increasing order of global rank, breaking ties randomly, to obtain a list L^* which is the aggregation of all top- k lists L_i -s.

Example 5.1. Figure 8 shows three top- k lists L_1, L_2, L_3 and the global list L^* . A line such as “a: 1.0 (1)” under L_1 means that item “a” in list L_1 has score 1.0 and has been assigned rank 1. The ranks for “a” is 1, 1, 2 (see Figure 8). So its global rank is 1. The ranks for “b” is 2, 4, 1 (here “b” is missing from L_2 , which has ranks 1-3; so we assign to it rank 4). Thus “b”’s global rank is 2. And so on.

Once we have obtained the global list L^* , we can present the top- n items of L^* to user U . But how do we incorporate the user feedback for the next iteration? A reasonable solution is to use weighted median ranking (WMR): we first assign an equal weight $w_i = 1/m$ to each top- k list L_i ($i \in [1, m]$). At the end of the first iteration, we adjust $w_i = w_i \cdot [1 + \log(1 + r_i)]$, where r_i is the number of true matches user U has identified that appear in L_i , then normalize all weights w_i . At the start of the next iteration, we merge the lists L_1, \dots, L_m again, using WMR to compute the global rank of each item. Next, we present the top- n pairs in this merged list to the user, and so on. Intuitively, the top- k lists in which more true matches appear will become more important, and the weighted global ranking will be “leaning toward” those lists.

Using Learning: WMR does not perform well in our experiments (see Section 6). It uses a very limited combination model

L_1	L_2	L_3	L^*
a: 1.0 (1)	a: 0.9 (1)	b: 0.8 (1)	a (1)
b: 0.8 (2)	c: 0.7 (2)	a: 0.5 (2)	b (2)
c: 0.8 (2)	d: 0.6 (3)	c: 0.3 (3)	c (2)
d: 0.6 (4)		d: 0.2 (4)	d (4)

Figure 8: Combining top- k lists using MedRank.

which fails to fully utilize user feedback. To address this, we explored active learning. Specifically, we iteratively show the next n items of L^* to user U , until we have obtained at least one match and one non-match. Suppose we have carried out t iterations, then this produces a set T of nt labeled items. We use T to train a random forest classifier F , use F to find n most informative items in L^* , show them to the user to label, add the newly labeled items to T , then retrain F , and so on.

Active learning alone however is not quite suited for our purpose. Its goal is to learn a good classifier as soon as possible. Hence it typically shows user U controversial items that it finds difficult to classify. But many or most of these items can be non-match. User U , however, wants to find many *true matches* as soon as possible (so that U can examine them to quickly understand the problems with the blocker).

The above two goals conflict. To address this problem, we adopt a hybrid solution. After we have obtained the training set T and trained a classifier F , as described above, for the next iteration, we show user U n items where $n/4$ items are the top controversial items chosen by F , as described above. The remaining $3n/4$ items however are those with the *highest positive prediction confidence*, where the confidence is computed as the fraction of decision trees in F that predict the item as a match. Intuitively, the first $n/4$ items are intended to help the active learner, whereas the remaining $3n/4$ items can contain many true matches, and are intended to help the user quickly find many true matches in the first few iterations.

After three such iterations, we stop active learning completely (judging that classifier F has received enough labeled controversial examples in order to do well), but continue the online-learning process with F . Specifically, in each subsequent iteration, we show user U the top n items with the highest positive prediction confidence, produced by F . Once these items have been labeled by U , we add them to the existing training set, retrain F , and so on.

When to Stop? A natural stopping point is when user U finds no new matches in 2 consecutive iterations. Of course, U can stop earlier or continue. If the required blocker recall is very high, U can continue for many iterations. Otherwise, U can stop after the first few iterations (because if these iterations contain many matches, then examining them often already reveals problems with the blocker, which U can then fix).

6 EMPIRICAL EVALUATION

We evaluated MatchCatcher in three ways. First, we experimented with a broad range of blockers that vary in recall, types, and complexity, representing blockers that users may write *at various points* during the blocker development process. We show that MatchCatcher works well with these blockers, thus can effectively support the users in the development process.

Second, we experimented with blockers that are either the best hash blockers manually developed or the best blockers automatically learned using a state-of-the-art solution. We show that even in these cases MatchCatcher can help uncover problems and improve the blockers.

Dataset	Tuple type	Table A	Table B	# of matches	# of attrs	Average length
Amazon-Google	software product	1363	3226	1300	5	205.38
Walmart-Amazon	electronic product	2554	22074	1154	7	76.179
ACM-DBLP	paper	2294	2616	2224	5	16.19
Fodors-Zagats	restaurant	533	331	112	7	11.10
Music ₁	song	100000	100000	2978	8	9.9
Music ₂	song	500000	500000	73646	8	9.9
Papers	paper	455996	628231	unknown	7	17.18

Table 1: Datasets for our experiments.

Dataset	Blocker Q
A-G	(OL) title_overlap_word<3 (HASH) attr_equal_manuf (SIM) title_cos_word<0.4 (R) title_jac_word<0.2 AND manuf_jac_3gram<0.4
W-A	(OL) title_overlap_word<3 (HASH) attr_equal_brand (SIM) title_cos_word<0.4 (R) price_absdiff>20 OR title_jac_word<0.5
A-D	(OL) authors_overlap_word<2 (SIM) title_jac_3gram<0.7 (R ₁) title_cos_word<0.8 AND authors_jac_3gram<0.8 (R ₂) year_abs_diff>0.5 OR title_jac_word<0.7
F-Z	(OL) name_overlap_word<2 (HASH) attr_equal_city (SIM) addr_jac_3gram<0.3 (R) (name_cos_word<0.5 AND type_jac_3gram<0.7) OR addr_jac_3gram<0.3
M ₁	(OL) artist_name_overlap_word<2 (HASH) attr_equal_artist_name (SIM) title_cos_word<0.5 (R) year_absdiff>0.5 OR title_cos_word<0.7
M ₂	(HASH ₁) attr_equal_artist_name (HASH ₂) attr_equal_release_OR_attr_equal_artist_name (SIM ₁) title_cos_word<0.6 (SIM ₂) title_cos_word<0.7 (SIM ₃) title_cos_word<0.8

Table 2: Blockers for the first set of experiments.

Finally, we asked real-world users in several data science classes, domain science projects, and at several organizations to use MatchCatcher. We show that MatchCatcher has proven highly effective in helping these users develop blockers.

6.1 Supporting Users in Developing Blockers

For this experiment we need “gold” matches, so we use the six datasets shown in the first six rows of Table 1. As far as we can tell, these datasets are the largest ones used in previous EM work for which “gold” matches are available. Here we created two versions of the Music dataset, Music1 and Music2, to ensure a diversity of size (from 331 to 100K to 500K of tuples per table). The technical report [24] describes these datasets in details.

For each dataset we asked volunteers to create multiple blockers (see Table 2). They are of the types described in Section 2: overlap (OL), hash (HASH), similarity-based (SIM), and rule-based (R). For example, the first row of Table 2 describes 4 blockers for dataset A-G. These include a hash blocker on attribute “manufacturer” and a rule-based blocker that combines two SIM blockers. See [24] for more details on these blockers. (The next subsection describes experiments with the best hash blockers manually created for these datasets.)

Developing a blocker is typically a *long process* in which users often start with a simple blocker, then revise it into more complex ones with higher recall. The above blockers differ in type, recall, and complexity, representing blockers that users may write at various points during the above process. We now show that MatchCatcher can help debug these blockers, suggesting that it can support the user during the entire development process.

Overall Accuracy: First we examine the top-k SSJs module. The first two columns of Table 3 list datasets and blockers. Column C lists the size of C , the output of the blocker on Tables A and B . Column M_D lists the number of true matches in $D = A \times B - C$. This number varies drastically, e.g., 137-1,267 for A-G, 87-566 for W-A, etc., suggesting that blocker recall often varies widely and that it is important to debug to improve recall.

Column E lists the size of E , the union of all top-k lists over the configs (for $k = 1000$). Column M_E lists the number of true matches in E (the numbers outside parentheses), and shows that set E contains a substantial fraction of true matches in D , e.g., 54-65% for A-G, 41-83% for W-A, 96-100% for A-D, etc. (see the

	Q	C	M_D	E	M_E	F	I
A-G	OL	8,388	291	4,063	190 (65.3)	166 (87.4)	40
	HASH	1,835	1,267	3,337	820 (64.7)	803 (97.9)	97
	SIM	7,406	192	4,341	104 (54.2)	73 (70.2)	29
	R	27,650	137	4,362	76 (55.5)	65 (85.5)	24
W-A	OL	210,782	87	6,570	48 (55.2)	37 (77.1)	7
	HASH	256,341	201	5,089	168 (83.6)	147 (87.5)	26
	SIM	46,900	135	7,089	56 (41.5)	46 (82.1)	7
	R	4,265	566	5,027	256 (45.2)	233 (91.0)	33
A-D	OL	56,869	41	4,270	41 (100.0)	37 (90.2)	8
	SIM	2,487	61	3,335	59 (96.7)	56 (94.9)	11
	R ₁	3,764	41	3,843	41 (100.0)	38 (92.7)	10
	R ₂	2,173	107	3,011	104 (97.2)	101 (97.1)	16
F-Z	OL	115	47	5,079	46 (97.9)	46 (100.0)	5
	HASH	10,165	52	4,653	51 (98.1)	51 (100.0)	5
	SIM	2,146	13	5,908	12 (92.3)	12 (100.0)	5
	R	124	33	5,239	32 (97.0)	32 (100.0)	5
M ₁	OL	253,286	778	5,045	673 (86.5)	671 (99.7)	38
	HASH	212,296	188	4,948	100 (53.2)	100 (100.0)	13
	SIM	2,601,349	78	5,050	38 (48.7)	36 (94.7)	7
	R	89,344	202	5,213	113 (55.9)	109 (96.5)	11
M ₂	HASH ₁	11,115,136	4,530	5,428	661 (14.6)	648 (98.0)	47
	HASH ₂	14,632,318	3,844	5,735	450 (11.7)	432 (96.0)	35
	SIM ₁	27,461,378	2,220	5,420	1,012 (45.6)	1,012 (100.0)	54
	SIM ₂	14,924,148	3,238	5,533	1,087 (33.6)	1,087 (100.0)	58
	SIM ₃	8,512,446	4,228	5,587	1,151 (27.2)	1,151 (100.0)	61

Table 3: Accuracy in retrieving the killed-off matches.

numbers in parentheses). This suggests that the top-k module can effectively find the true matches in D .

Next we examine the Match Verifier. We want to know its accuracy if run until its natural stopping point (see Section 5). It is difficult to recruit enough real users for this large-scale experiment involving 25 blockers. So we use synthetic users, whom we assume can identify the true matches accurately (we describe multiple experiments with real users below).

Column F of Table 3 show that this module can retrieve a large number of matches in E , e.g., 65-803 for A-G (see the numbers outside parentheses), and that the retrieval rate is very high, e.g., 70-98% for A-G, 77-91% for W-A, etc. (see the numbers inside parentheses). Finally, Column I shows that the total number of iterations is 5-13 in 12 cases, 16-40 in 8 cases, 47-61 in 4 cases, and 97 in 1 case. The higher number of iterations is often due to the larger number of matches that have to be retrieved from E , e.g., for blocker HASH of dataset A-G, the module needed 97 iterations to retrieve 803 matches, a reasonable number of iterations given that each iteration shows only 20 tuple pairs to the user.

Thus, if the user runs the Match Verifier until its natural stopping point, he/she can retrieve a large number of matches. This is good news for applications in which blocker recall is critical, thus the user may want to examine *all* matches that the module can retrieve.

Accuracy & Explanations for the First Few Iterations: To examine if users can quickly find many matches and explanations, we asked volunteers to *manually work with the Match Verifier for the first three iterations*. Table 4 shows the results (for space reasons we only list five blockers for five datasets, the results for other blockers are similar). The table shows that the user needed only 7-10 mins to examine the first three iterations, was able to identify a large number of matches (28-43), and was able to identify multiple reasons for why they are killed off (a reason such as “large threshold (18)” means that tuple pair #18 was killed off due to the blocker using a large threshold, and this was the first pair where the user observed this problem). Overall, the results suggest that after examining the first few iterations, the

Blocker	# iteration	Label time	Blocker problems
OL (A-G)	3 iterations 31 matches	8 mins	large threshold (18); attribute "manuf" is sprinkled in the attribute "title" (18)
HASH (W-A)	3 iterations 43 matches	10 mins	different words for the same brand (6); missing values in attribute "brand" (13)
SIM (A-D)	3 iterations 28 matches	7 mins	large threshold (16); attribute "title" contains subtitle in one table (22)
R (F-Z)	3 iterations 32 matches	7 mins	different descriptions for attribute "type" (11); unnormalized attribute "address" (33); attribute "city" is sprinkled in "name" (47)
R (M ₁)	3 iterations 41 matches	10 mins	input tables are not lower-cased (5); missing values in attribute "year" (12)

Table 4: Accuracy in the first 3 iterations and explanations. user can already identify multiple problems with the blocker (which he/she can then fix).

6.2 Debugging State-of-the-Art Blockers

Suppose a user has manually developed a good standard blocker, or has used state-of-the-art techniques to learn a blocker, we want to know if MatchCatcher can still help improve the blocker’s accuracy. Toward this goal, we performed two experiments.

Hash Blockers: First, we asked a user well-trained in EM to develop the best possible hash blockers for five datasets (the first five in Table 1). For example, for dataset A-G, this user created the blocker Q_1 which keeps a pair of tuples if they agree on “manufacturer” or on a hash of “price” or on a hash of “title”. Thus, Q_1 combines three hash blockers. ([24] describes all five blockers in details.) We selected hash blocking because it is well-known, easy to understand, and fast. Hence it is considered a standard blocking method commonly used in practice. On the five datasets A-G, W-A, A-D, F-Z, and Music1, the best hash blockers achieve 75.6, 95.1, 100, 97.3, and 100% recall, respectively.

We then asked the same user to use MatchCatcher to try improving the above hash blockers. For A-D and Music1, which already have 100% recall, using MatchCatcher the user did not find any killed-off matches (as expected), so debugging terminated early. For A-G, W-A, and F-Z, however, debugging significantly improved recall from 75.6 to 99.7, 95.1 to 99.6, and 97.3 to 100%, respectively. [24] describes one such debugging scenario in details.

Learned Blockers: From a group of researchers we obtained Papers, the dataset described in the last row of Table 1. For this dataset, they have applied the method in [8] to learn blockers using a sample labeled by crowdsourcing, and we were able to obtain three such blockers (learned on three separate samples). The technical report [24] describes these blockers, which are the best blockers that the learning method has found in a very large space of blockers, including hash ones. Unfortunately, we do not have the entire set of “gold” matches for Papers (we do have some “gold” matches, but not all of them). Hence, we are unable to report recalls for these blockers.

We then asked a user to apply MatchCatcher to these blockers. After 5 iterations, the user found 76, 61, and 65 matches for the three blockers, respectively. More importantly, the user was able to identify a set of reasons for why these matches were killed off and suggestions for improving the blockers (see [24]). Given the lack of “gold” matches, we were not able to improve the blockers then compare their recalls. Nevertheless, the above experiments suggest that blockers learned using state-of-the-art solutions can still have many problems and MatchCatcher can help pinpoint these, to help the user improve recall.

6.3 MatchCatcher “in the Wild”

Over the past two years variations of MatchCatcher have been used by 300+ students in 4 data science classes and 7 EM teams at

6 organizations. The feedback has been overwhelmingly positive. For example, 18 teams used MatchCatcher in a class project, and reported that it helped (a) discovering data that should be cleaned, (b) finding the correct blocker types and attributes, (c) tuning blocker parameters, and (d) knowing when to stop. We have reported on some of this experience in [23]. Overall, we found that many real-world users have used MatchCatcher as *an integral part of an end-to-end blocker development process*: start with a simple blocker, use MatchCatcher to identify problems, improve the blocker, and so on, until MatchCatcher no longer reports substantial problems with the blocker.

6.4 Runtime & Scalability

MatchCatcher was implemented in Cython, and all experiments used a RedHat 7.2 Linux machine with Intel E5-1650 CPU. The top-k module took 6.6-9.4 secs (for dataset A-G), 97-310 (W-A), 2.8-3.2 (A-D), 0.2 (F-Z), 12.1-24.4 (M₁), 57-230 (M₂), and 65-344 (Papers), respectively. For the first five datasets, these times are quite small except 97-310 secs for W-A. On W-A, the k-th pair on the top-k list (recall that $k = 1000$) often has a very low score, e.g., 0.21-0.225. Thus the top-k module took more time. The last two datasets (M₂ and Papers) are much larger (500K tuples per table), and so took longer to run. In all cases, however, the total time is still under 5.8 minutes.

To examine how the top-k module scales, we measure its time as we vary the size of the two largest datasets, M₂ and Papers, at various percentages of the original datasets (which have 500-600K tuples per table). Figure 9 shows the results for the first three blockers in Table 3 for M₂ and all three blockers for Papers, for $k = 100$ (the left two plots) and $k = 1000$. The results show that the top-k module scales linearly or sublinearly as the table size grows. Finally, on all datasets the Match Verifier took under 0.1 sec to aggregate the top-k lists, and 0.14-0.18 secs to process user feedback in each iteration.

6.5 Additional Experiments

The technical report [24] describe extensive experiments on the performance of the MatchCatcher components, sensitivity analysis, and comparison with a recent related work. For space reasons we only briefly summarize those experiments here.

Performance of the Components: We show that using multiple configs instead of just one config significantly increases the number of retrieved matches, by 10-74%. Handling long attributes increases the recall of E (the fraction of matches in D that are in E) by up to 11%, compared to not handling them in config generation. Our experiments also show that the joint top-k processing strategy over multiple configs significantly outperforms the baseline of executing each config individually, by as much as 3.5 times. Finally, we found that active/online learning significantly outperforms weighted median ranking in the Match Verifier.

Sensitivity Analysis: We found that increasing k (the number of pairs retrieved per config) does increase the number of true matches retrieved, but only up to a certain k , and comes at the cost of higher runtime, and that using 3 active learning iterations (as we currently do) provides a good balance between increasing the classifier accuracy and increasing recall in the Match Verifier.

Comparison with Recent Work: We found MatchCatcher significantly outperforms the work in [29], which uses a single config, e.g., improving the recall of E by 26-47% on the A-G dataset.

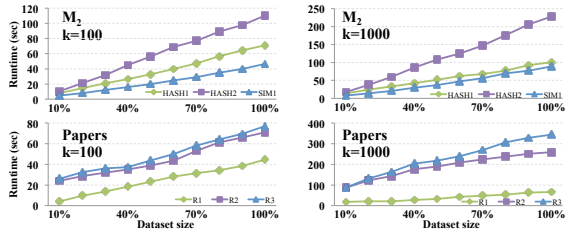


Figure 9: Runtime of top-k module for varying table sizes.

7 ADDITIONAL RELATED WORK

We have discussed related work throughout the paper. We now discuss additional related work. As far as we can tell, our recent work [23] is the first to raise the need for debugging for blocking. But that work focuses on developing end-to-end EM systems. It does not discuss any debugging solution in depth, as we do here. Other related works include debugging for data cleaning [17], schema mapping [5], and data errors in spreadsheets [1]. They do not address EM and their solutions do not apply to our context. But they do underscore the importance of debugging for data integration and cleaning.

SSJs have received much attention, e.g., [21, 35] (see [36] for a survey). Top-k SSJs are studied in [34, 37]. [37] proposes a B+ tree based index to scale top-K SSJs on edit distance. It does not work well for datasets with large textual difference [36], however, a common occurrence in our case. The work [34], which uses prefix filtering to find the top-k pairs, is better suited to our case. But it does not handle long strings well [36]. Here we have significantly improved this work and extended it to work over multiple configs.

The idea of computing the similarity score of a string pair only if their prefixes share at least q tokens (see Algorithm QJoin in Section 4.1) is also discussed in [32]. That work however focuses on SSJs with thresholding, e.g., matching two strings x and y if $jaccard(3g(x), 3g(y)) \geq \alpha$. Its solution uses threshold α to find the optimal q , and is not applicable to the top-k context considered in this paper (which has no threshold α).

The work [27] describes a blocking method that performs a variation of weighted overlap blocking to find tuples that are highly similar string-wise. MatchCatcher however does not use this method in top-k SSJs because it is not clear how to modify it to enable reuse (among the different configs). The work [16] is related to our work on config generation in that it defines the notion of matching dependencies, using which we can deduce a set of attributes for comparing tuple pairs. However, it is not applicable to our context because it requires the user to manually specify matching dependencies, using domain knowledge, in a potentially time consuming process.

Rank aggregation has been studied extensively in the database/IR communities, e.g., [4, 11, 15]. Active learning (AL) for EM has been studied in [18, 26, 28]. But they perform extensive AL to learn an accurate matcher. In contrast, we use only a few AL iterations to learn a classifier with reasonable accuracy, then use it to surface matches for debugging purposes. The above work also does not combine AL with online learning as we do. Finally, the work [31] uses a learning-based UI model similar to ours, but for IR tasks.

8 CONCLUSIONS & FUTURE WORK

We have shown that debugging blocker accuracy is critical for EM, and have described MatchCatcher, a solution to this problem.

As for future work, in certain cases the user may find a large number of killed-off matches. So we plan to develop a method to automatically explain why each match is killed off by the blocker, summarize these explanations, then present the summary to the user. When fixing a problem affecting a killed-off match, the user may want to know how pervasive this problem is (and focus on fixing the most pervasive ones first). For this purpose, given a killed-off match, we plan to develop a method to find all tuple pairs that are similar to that match (from a blocking point of view).

REFERENCES

- [1] D. Barowy, D. Gochev, and E. Berger. 2014. CheckCell: data debugging for spreadsheets. OOPSLA.
- [2] M. Bilenco, B. Kamath, and R. J. Mooney. 2006. Adaptive blocking: learning to scale up record linkage. ICDE.
- [3] M. Bilenco and R. J. Mooney. 2003. Adaptive duplicate detection using learnable string similarity measures. SIGKDD.
- [4] B. Brancotte, B. Yang, G. Blin, S. Cohen-Boulakia, A. Denise, and S. Hamel. 2015. Rank aggregation with ties: experiments and analysis. VLDB.
- [5] L. Chiticariu et al. 2006. Debugging schema mappings with routes. VLDB.
- [6] P. Christen. 2012. *Data Matching*. Springer.
- [7] P. Christen. 2012. A survey of indexing techniques for scalable record linkage and deduplication. *IEEE TKDE* 24, 9 (2012), 1537–1555.
- [8] S. Das et al. 2017. Falcon: scaling up hands-off crowdsourced entity matching to build cloud services. SIGMOD.
- [9] A. Doan, A. Halevy, and Z. Ives. 2012. *Principles of Data Integration*. Elsevier.
- [10] X. Dong, A. Halevy, and J. Madhavan. 2005. Reference reconciliation in complex information spaces. SIGMOD.
- [11] C. Dwork et al. 2001. Rank aggregation methods for the web. WWW.
- [12] V. Efthymiou, G. Papadakis, et al. 2017. Parallel meta-blocking for scaling entity resolution over big heterogeneous data. *Inf. Syst.* 65 (2017), 137–157.
- [13] V. Efthymiou, K. Stefanidis, and V. Christophides. 2016. Benchmarking blocking algorithms for web entities. *IEEE Transactions on Big Data* (2016).
- [14] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. 2007. Duplicate record detection: a survey. *IEEE Trans. Knowl. Data Eng.* 19, 1 (2007), 1–16.
- [15] R. Pagan, R. Kumar, and D. Sivakumar. 2003. Efficient similarity search and classification via rank aggregation. SIGMOD.
- [16] W. Fan et al. 2009. Reasoning about record matching rules. VLDB.
- [17] H. Galhardas, D. Florescu, D. Shasha, et al. 2000. AJAX: an extensible data cleaning tool. SIGMOD.
- [18] C. Gokhale, S. Das, A. Doan, J. F. Naughton, N. Rampalli, J. Shavlik, and X. Zhu. 2014. Corleone: hands-off crowdsourcing for entity matching. SIGMOD.
- [19] Y. Govind et al. 2017. CloudMatcher: A Cloud/Crowd Service for Entity Matching. BIGDAS@KDD.
- [20] M. A. Hernández and S. J. Stolfo. 1998. Real-world data is dirty: data cleansing and the merge/purge problem. *Data Min. Knowl. Discov.* 2, 1 (1998), 9–37.
- [21] Y. Jiang, G. Li, J. Feng, et al. 2014. String similarity joins: an experimental evaluation. VLDB.
- [22] L. Kolb, A. Thor, and E. Rahm. 2011. Parallel sorted neighborhood blocking with MapReduce. BTW.
- [23] P. Konda et al. 2016. Magellan: toward building entity matching management systems. VLDB.
- [24] H. Li et al. 2017. *MatchCatcher: a debugger for blocking in entity matching*. Technical Report. <http://pages.cs.wisc.edu/~anhai/papers1/matchcatcher-tr.pdf>.
- [25] M. Michelson. 2006. Learning blocking schemes for record linkage. AAAI.
- [26] B. Mozafari, P. Sarkar, M. Franklin, M. Jordan, and S. Madden. 2014. Scaling up crowd-sourcing to very large datasets: a case for active learning. VLDB.
- [27] G. Papadakis, G. Koutrika, T. Palpanas, and W. Nejdl. 2014. Meta-blocking: taking entity resolution to the next level. *IEEE TKDE* 26, 8 (2014), 1946–1960.
- [28] S. Sarawagi and A. Bhamidipaty. 2002. Interactive deduplication using active learning. SIGKDD.
- [29] D. Song and J. Heflin. 2011. Automatically generating data linkages using a domain-independent candidate selection approach. ISWC.
- [30] K. Stefanidis, V. Efthymiou, M. Herschel, and V. Christophides. 2014. Entity resolution in the web of data. WWW (Companion Volume).
- [31] A. Tian and M. Lease. 2011. Active learning to maximize accuracy vs. effort in interactive information retrieval. SIGIR.
- [32] J. Wang, G. Li, and J. Feng. 2012. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. SIGMOD.
- [33] S. E. Whang, D. Menestrina, G. Koutrika, M. Theobald, and H. Garcia-Molina. 2009. Entity resolution with iterative blocking. SIGMOD.
- [34] C. Xiao, W. Wang, X. Lin, and H. Shang. 2009. Top-k set similarity joins. ICDE.
- [35] C. Xiao, W. Wang, X. Lin, J. X. Yu, and G. Wang. 2011. Efficient similarity joins for near-duplicate detection. *TODS* 36, 3 (2011), 15.
- [36] M. Yu, G. Li, D. Deng, and J. Feng. 2016. String similarity search and join: a survey. *Frontiers of Computer Science* 10, 3 (2016), 399–417.
- [37] Z. Zhang, M. Hadjieleftheriou, B. Ooi, et al. 2010. Bed-tree: an all-purpose index structure for string similarity search based on edit distance. SIGMOD.

Extracting Statistical Graph Features for Accurate and Efficient Time Series Classification

Daoyuan Li
University of Luxembourg
Luxembourg City, Luxembourg
daoyuan.li@uni.lu

Jessica Lin
George Mason University
Fairfax, Virginia, U.S.A.
jessica@gmu.edu

Tegawendé F. Bissyandé
University of Luxembourg
Luxembourg City, Luxembourg
tegawende.bissyande@uni.lu

Jacques Klein
University of Luxembourg
Luxembourg City, Luxembourg
jacques.klein@uni.lu

Yves Le Traon
University of Luxembourg
Luxembourg City, Luxembourg
yves.letraon@uni.lu

ABSTRACT

This paper presents a multiscale visibility graph representation for time series as well as feature extraction methods for time series classification (TSC). Unlike traditional TSC approaches that seek to find global similarities in time series databases (e.g., Nearest Neighbor with Dynamic Time Warping distance) or methods specializing in locating local patterns/subsequences (e.g., shapelets), we extract solely statistical features from graphs that are generated from time series. Specifically, we augment time series by means of their multiscale approximations, which are further transformed into a set of visibility graphs. After extracting probability distributions of small motifs, density, assortativity, etc., these features are used for building highly accurate classification models using generic classifiers (e.g., Support Vector Machine and eXtreme Gradient Boosting). Thanks to the way how we transform time series into graphs and extract features from them, we are able to capture both global and local features from time series. Based on extensive experiments on a large number of open datasets and comparison with five state-of-the-art TSC algorithms, our approach is shown to be both accurate and efficient: it is more accurate than Learning Shapelets and at the same time faster than Fast Shapelets.

1 INTRODUCTION

Time series data refer to sequences of data that are ordered either temporally, spatially or in another defined order. They can be frequently found in a variety of domains, including financial data analysis, medical and health monitoring and industrial automation applications. Recently, it turns out to be feasible to model software systems as time series in order to conduct malware detection and classification [40]. Due to their wide application scenarios and abundance, there has been an increasing need for efficient knowledge discovery methods to extract useful information from time series databases. One of the major tasks in time series mining is time series classification (TSC), which consists of applying a learning algorithm on labeled data to train a model that will then be used to predict the classes of samples from an unlabeled data set. Due to the sequential characteristic of time series data, state-of-the-art classification algorithms (such as SVM and Random Forest [13]) that perform well for generic data are generally not suitable for TSC. It is thus important and beneficial

to have a feature extraction mechanism that transforms the sequential characteristics of time series data into unordered feature vectors, so that any modern classification algorithm can be taken advantage of. After all, one of the most challenging aspects of TSC lies in the sequentiality property.

Traditionally, researchers often rely on one of the simplest classifiers for TSC: the k Nearest Neighbor (k NN) algorithm. As stated in [6], “all of the current empirical evidence suggests that simple nearest neighbor classification is very difficult to beat”. To perform well, k NN classifiers leverage the Dynamic Time Warping (DTW) [7] distance which mitigates problems caused by distortion in the time axis. One intrinsic issue with DTW, however, is that it focuses on finding *global* similarities, i.e., the overall curve *shape* of time series. It also requires applications to specify a proper warping window size or to properly align data samples. As a result, DTW can be sensitive to data alignment/segmentation and it performs better when data are properly curated. In practice, however, well-aligned time series data are difficult or expensive to come by [17].

To address the phase issue of DTW- and other distance-based 1NN approaches, the research community has proposed approaches that focuses on finding defining *local* features/subsequences in order to be invariant to data alignment and rotation. Popular methods that fall into this category include Bag-of-Patterns [27], SAX-VSM [36] and shapelets-based algorithms [44], such as Fast Shapelets (FS) [31] and Learning Shapelets (LS) [15]. The majority of these techniques have taken advantage of text feature extraction approaches – e.g., TF-IDF – after converting time series into alphabetical strings. Such conversion is often done via Symbolic Aggregate approxImation (SAX) [26], which requires two parameters (i.e., cardinality and PAA window size) to be set and it may not always be trivial to find the best pair. Besides, many approaches attempt to find time series subsequences that are representative of each class, e.g., shapelets by definition are defining time series subsequences that are calculated by exhaustive or optimized search. Overall, many of these methods have suffered from high computation complexity or suboptimal classification accuracy [39].

Graph representations for TSC, on the other hand, have not been investigated extensively by the time series mining research community possibly due to their high computation complexity. Nevertheless, thanks to recent development of graph mining algorithms [5, 29], some of the formerly complex problems can be solved extremely efficiently with optimization and parallelization techniques [2]. Such advances give us the opportunity to

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

re-evaluate the possibility of taking advantage of graph representations and extracting graph features for building an efficient and accurate TSC algorithm.

This paper proposes a novel approach for TSC that considers time series as complex graphs/networks and extracts from these networks important statistical features, which are fed to modern generic classifiers to learn structural knowledge from the original time series. After evaluating the classification performance with a large open dataset, we find out that our approach is capable of making efficient and accurate classification predictions. The main contributions of this paper are listed as follows:

- We present a multiscale graph representation for time series, so that both global and local features from time series can be captured, making this approach agnostic to time series alignment and outperform major distance-based TSC algorithms.
- Since we transform time series that are intrinsically sequential into unordered feature vectors, it is then suitable for taking advantage of modern generic classifiers (*e.g.*, RF, SVM and XGBoost) for efficient feature selection and classification. This clear separation of feature extraction and actual classification can help researcher focus on finding insightful characteristics in time series data without the need for reinventing the wheel and designing a classifier from scratch specifically for time series.
- We propose a novel feature extraction and classification method for time series based on calculating probability distributions of small motifs (*i.e.*, repeated patterns) in visibility graphs and other statistical features such as density, degree statistics, assortativity and coreness. This feature extraction mechanism is parameter-free, so that it can be easy to use and help yield reproducible results.
- We have intentionally chosen a collection of statistical features that are computationally efficient to extract from graphs and validated their effectiveness in controlled experiments. Moreover, since our feature extraction and classification process is inherently parallel, it is suitable for and capable of large scale data explorations.
- After extensively evaluating our approach with a large number of open datasets and comparing with related research efforts, experiment results indeed suggest that accurate and efficient classification can be obtained following this paradigm.

The remainder of this paper is structured as follows. Section 2 lays down the necessary background. Next, we present our approach in section 3 and detail TSC accuracy and efficiency evaluation results along with a case study in section 4. For interested readers, section 5 introduces research work related to ours. Finally, we conclude the paper with future research directions in section 6.

2 BACKGROUND

Traditionally, time series refer to a sequence of numbers that are chronologically ordered. However, in the research community time series have a much broader scope and do not associate strictly with timestamps:

Definition 2.1 (Time series). A time series instance T is an ordered sequence of n real-valued variables, *i.e.*, $T = (v_1, \dots, v_n), v_i \in \mathbb{R}$.

If we consider each point in a time series as a single feature in a vector, then time series data usually have a huge number

of features. When considering these features as a vector in an n -dimensional space, time series data are often high dimensional. Due to difficulties to conduct knowledge discovery tasks on high dimensional data, it is frequently required to reduce the dimensionality of time series in order to improve computation efficiency:

Definition 2.2 (Dimensionality reduction). The dimensionality of a time series sample T is the length of T , denoted by $|T|$. If T' is an approximated representation of T and $|T'| \ll |T|$, then T' is a dimension-reduced representation of T .

The research community has proposed a number of dimensionality reduction techniques for time series, including sampling [32], Piecewise Linear Representation (PLR) [19], Piecewise Aggregate Approximation (PAA) [20], *etc.*. Among them PAA is perhaps one of the simplest and most widely applied approaches. PAA reduces time series $T = (v_1, \dots, v_n)$ from n dimensions to s dimensions by firstly dividing the data into s segments of equal size, then the approximation is a vector of the mean values of the data readings per segment [20, 26]. Let $T' = (v'_1, \dots, v'_s)$ be this vector where v'_i is computed by equation 1. For the sake of simplicity, $\frac{n}{s}$ is often chosen to be an integer or rounded to the nearest one.

$$v'_i = \frac{s}{n} \sum_{k=\frac{n}{s}(i-1)+1}^{\frac{n}{s}i} v_k \quad (1)$$

One of the core routines in distance-based classification algorithms involves evaluating the dissimilarity (or similarity) of two time series. There are a number of dissimilarity measures for time series, two of the most frequently used measures in the research community are Euclidean distance and DTW distance. The Euclidean distance maintains a one-to-one mapping of all the points in two series. On the other hand, the DTW distance tries to find the best mapping of points in two series using the dynamic programming paradigm, so that the minimum distance between these two series is achieved. The paradigm is called “time warping” since the time axis of series can be expanded or compressed in order to ensure the minimum distance, *i.e.*, an i^{th} point in X can be mapped to a j^{th} point (it is possible that $i \neq j$), or one point in X may even be mapped to multiple points in Y .

2.1 Visibility Graph

Aiming for taking advantage of graph theories as a way of characterizing time series, an algorithm named visibility graph (VG) [23, 24] is proposed to transform time series into a network structure. The creation of VGs relies on an extremely simple idea: each point in a time series is treated as a vertical bar, whose height is the corresponding numerical value. When considering these bars on a landscape, it is then straightforward that the top of a bar may be visible from the top of other bars. Assume each time-step as a vertex in a graph, then two vertices are connected if the top of the vertical bars are visible to each other, *i.e.*, there exists a straight line from the top of the two bars without intersecting with other bars. More formally,

Definition 2.3 (Visibility graph). Given a time series $T = (v_1, \dots, v_n)$, its VG representation $G = (V, E)$ has n vertices: $V = (1, \dots, n)$. An edge $e = (i, j) \in E$ iff $\forall k$ such that $i < k < j$ ($1 \leq i, j \leq n$) inequality $v_k < v_j + (v_i - v_j) \frac{j-k}{j-i}$ is satisfied.

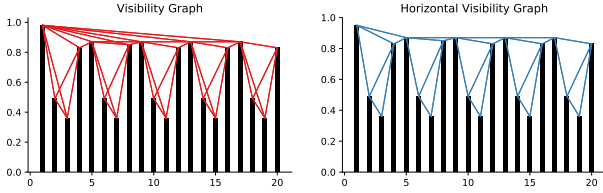


Figure 1: An example of converting time series to visibility graph and horizontal visibility graph.

It is obvious that VGs are undirected, although it is possible to create a directed version by limiting the direction of view-points from the vertical bars. Besides, VGs are always connected since each node will always be visible to its neighbors. Finally, VGs are invariant when time series undergo affine transformations, *i.e.*, the visibility criterion remains fulfilled when rescaling the time series either horizontally or vertically. However, VGs are not suitable for non-stationary time series, *i.e.*, those having monotonically increasing/decreasing trends in time. Such trends should be removed before applying VG generation. The goal of VGs is to characterize structural properties [24] of time series, such as periodicity, fractality, *etc.*, although it is shown that it can be extended to weighted VGs in order to quantitatively distinguish generic time series [38].

Creation of VGs from time series without optimization generally has $O(n^2)$ computation complexity, where n is the dimensionality of time series data. However, a more efficient VG generation algorithm [1] can have a sub-quadratic computation complexity. Specifically, this algorithm can reduce the complexity of time series VG generation down to $O(n \log^2(n))$. When further taking advantage of parallelization, $O(n \log^2(n))$ work can be effectively solved within $O(\log^2(n))$ time. Another simplified variant of VG, horizontal visibility graph (HVG) [28], only connects nodes i and j if a horizontal line can be drawn between these nodes. Creation of HVGs without any optimization generally has a computation complexity of $O(n)$.

Definition 2.4 (Horizontal visibility graph). Given a time series $T = (v_1, \dots, v_n)$, its HVG representation $\hat{G} = (V, E)$ has n vertices: $V = (1, \dots, n)$. An edge $e = (i, j) \in E$ iff $\forall k$ such that $i < k < j$ ($1 \leq i, j \leq n$) inequality $v_i, v_j > v_k$ is satisfied.

VG and its variants have been shown to be able to differentiate certain time series. For instance, [18] have extracted motifs from HVGs and claimed the motif statistics can be used for differentiating various types of time series data, including white Gaussian noises, fully chaotic logistic maps and noisy fully chaotic logistic maps. The authors further claimed that HVG motif profiles from heart rate time series can be used to cluster different types of meditative activities.

Intuitively, VGs and HVGs are very similar concepts and HVG is a subgraph of VG. However, when extracting statistics from them, VGs can be more capable of capturing global features, while HVGs are often more sensitive to local variations. As a result, VGs and HVGs can be joined to provide more accurate representations of time series data. We will discuss more about such heuristics in section 4.2. For simplicity, in the remainder of this paper the term VG indicates the combination of VG and HVG if HVG is not explicitly specified.

2.2 Graph Features

Extracting features from graphs has become a popular research topic thanks to the recent applications in social network analysis, physics as well as bio-informatics. There are a number of

research avenues in graph mining, the most popular ones are about finding communities or clusters within large graphs and characterizing graphs by means of finding and counting recurrent patterns. Since time series VGs are always connected, it is thus not immediately helpful to extract clusters, since clusters in these graphs will always correspond to subsequences of original time series. Graph motifs or graphlets, however, are especially interesting since they are sub-graph structures or patterns in a larger graph that are recurrent and statistically significant. Finding motifs in graphs is generally a complex problem, but there have been a number of research work for efficiently extracting motifs from graph data, such as GTrieScanner [34] and Parallel Parameterized Graphlet Decomposition (PGD) [2]. Due to the fact that the number of motifs increases exponentially with motif size, the complexity of finding motifs is also exponentially correlated with motif size. Researchers thus generally focus on optimizing computation efficiency of locating small motifs (of size up to four). PGD is the state-of-the-art approach for efficiently finding and counting small motifs in graphs and can be several magnitudes faster than other approaches. Table 1 shows all possible small graph motifs up to size four. Note that PGD works only for undirected graphs, while GTrieScanner works on both directed and undirected graphs. However, motifs extracted by GTrieScanner are only connected motifs and the running time is significantly slower than PGD. As a result, in this paper we take advantage of PGD for small motif counting.

Table 1: All graph motifs up to size 4. Note that connected graphs may contain disconnected motifs.

#	Motif	Name	#	Motif	Name
M_{21}		2-edge	M_{22}		2-node-independent
M_{31}		3-triangle	M_{33}		3-node-1-edge
M_{32}		3-path	M_{34}		3-node-independent
M_{41}		4-clique	M_{47}		4-node-triangle
M_{42}		4-chordal-cycle	M_{48}		4-node-star
M_{43}		4-tailed-triangle	M_{49}		4-node-2-edges
M_{44}		4-cycle	M_{410}		4-node-1-edge
M_{45}		4-star	M_{411}		4-node-independent
M_{46}		4-path			

Researchers argue that motif distribution extracted from VGs can be extremely helpful for identifying different types of artificially generated time series. However, in practice motif distributions may not be as distinguishable as those from artificial data. For example, Figure 2 illustrates the motif distributions of three different classes of time series from the ArrowHead dataset. As shown, it can be very difficult to tell one class apart from another given only these distributions since the probability distributions of different classes tend to overlap, especially in this case for instances from class 2 and 3. Besides small motifs, other graph features are also easy to obtain, *e.g.*, vertex and edge statistics as well as structural metrics. In this paper we consider some additional graph features listed as follows:

- Density: the ratio of the number of edges to the number of all possible edges. Graph density is computed following equation 2 and has a computation complexity of $O(1)$.

$$p = \frac{2|E|}{|V|(|V| - 1)} \quad (2)$$

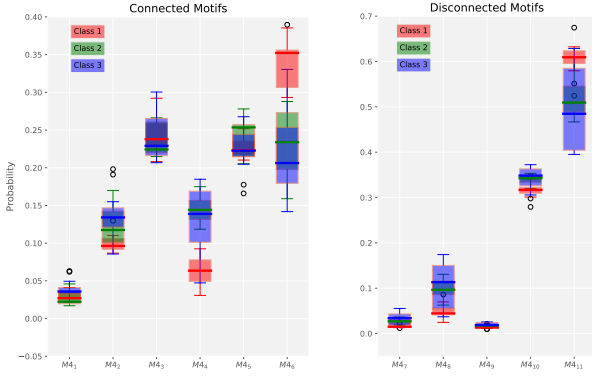


Figure 2: Boxplots of motif probability distribution of different classes from the ArrowHead Dataset’s training set.

- *K*-core: the *K*-core of a graph $G = (V, E)$ is a maximal subgraph $H = (V', E')$ in which each vertex has a degree of at least *K*, i.e., $\forall v \in V' : \deg_H(v) \geq K$. Consequently, it is a cohesiveness measurement of interlinked subgraphs within a network and *K* is computed by equation 3. Computing *K* has an $O(|E|)$ time complexity [5].

$$K = \operatorname{argmax}_{H \in G} \operatorname{Core}_H \geq K \quad (3)$$

- Assortativity coefficient: a metric to measure the correlation of vertices in a graph through calculating the Pearson correlation coefficient of degree between pairs of connected vertices. Equation 4 shows how assortativity coefficient is computed,

$$r = \frac{\sum_{xy} xy(e_{xy} - a_x b_y)}{\sigma_a \sigma_b} \quad (4)$$

where a_x and b_y represents respectively the fraction of edges that start and end at vertices with values x and y , and e_{xy} is the measure of assortativity, such that $\sum_{xy} e_{xy} = 1$, $\sum_y e_{xy} = a_x$ and $\sum_x e_{xy} = b_y$. Finally, σ_a and σ_b are the standard deviations of the distributions a_x and b_y . Computing r has an $O(|E|)$ time complexity [29].

- Degree statistics including max, min and mean degree per vertex. Naturally, computing such statistics has an $O(|V|)$ time complexity.

Note that there are a plethora of features that can be extracted from graphs [11], and such features are not necessarily equally easy to obtain. For instance, the diameter of a graph – which is the shortest distance between the two most distant vertices in the graph – can be computationally expensive to calculate since it demands $O(|V|(|V| + |E|))$ computation. Nonetheless, this paper does not intend to find the exclusive set of features that are both efficient and descriptive for time series VGs. Rather, we try to investigate if these aforementioned statistical features are indeed helpful for TSC.

3 MULTISCALE VISIBILITY GRAPH

Time series data differ greatly in characteristics depending on how they are captured, sampled and their underlying applications: in one domain global features may be helpful, while in another domain local features (i.e., defining subsequences) can become more important for classification. After transforming real-valued time series into VGs, specific features from such VGs – such as probability distributions of small motifs – are more reflective of local features than global ones. We refer to this type of

representation as Uniscale Visibility Graph (UVG) in the remainder of this paper. In this case, extracting more global features (i.e., probability distributions of large motifs) becomes exponentially expensive for computation. To mitigate this problem, we propose the Multiscale Visibility Graph (MVG) representation such that each time series sequence is transformed into a set of dimensionality-reduced approximations: these downscaled approximations are then converted into a set of VGs. Consequently, features are extracted from each graph in MVGs, thus approximating the process of extracting features of different scales. This approach is inspired by computationally efficient wavelet transform [10], with the exception that time series in each scale are represented with graph structures instead of numerical values. We first define the multiscale representation of time series as follows:

Definition 3.1 (Multiscale Approximation). Given a time series $T_0 = (v_1, \dots, v_n)$, its approximated multiscale representation is a set of time series $\hat{\mathbb{T}} = (T_1, T_2, \dots, T_m)$, where T_i ($1 \leq i \leq m$) is a downscaled approximation of T_0 , such that $|T_i| = |T_0|/2^i = n/2^i$. In addition, to avoid tiny and meaningless representations we enforce a constant threshold τ for the downscaled approximations, i.e., $|T_m| > \tau$.

The downscaling of T_0 can be achieved with widely used dimensionality reduction techniques such as PAA (cf. equation 1). Since the size of downscaled time series representations follows an exponential decay, time series multiscale representations $\hat{\mathbb{T}}$ often consists of a small number downscaled series. Besides, considering $\sum_{i=1}^{\infty} n * 2^{-i} = n$, theoretically the maximal dimension of $\hat{\mathbb{T}}$ when fully expanded is n .

Definition 3.2 (Multiscale representation). Given a time series T_0 and its approximated multiscale representation $\hat{\mathbb{T}} = (T_1, T_2, \dots, T_m)$, its multiscale representation is the union of $\hat{\mathbb{T}}$ and T_0 , i.e., $\mathbb{T} = (T_0, T_1, T_2, \dots, T_m)$.

Approximated multiscale representations can help smoothing time series and reducing noises, while full multiscale representations consist of both the original time series and augmented versions. Each series in multiscale representations can be transformed into VGs, thus we have a natural definition for multiscale visibility graphs, which are supersets of approximated multiscale visibility graphs (AMVGs):

Definition 3.3 (Multiscale visibility graph). Given a time series T and its multiscale representation $\mathbb{T} = (T_0, T_1, T_2, \dots, T_m)$, its multiscale visibility graph is a set of graphs $\mathbb{G} = (G_0, G_1, G_2, \dots, G_m)$, where G_i ($1 \leq i \leq m$) is the corresponding visibility graph created from T_i .

Since the number of vertices in G_i equals the dimensionality of T_i , to avoid meaningless trivial graphs it is natural to set τ to a small integer (e.g., $\tau = 15$), such that the smallest graph in \mathbb{G} contains more than τ vertices. Note that τ should not be considered as a parameter for the feature extraction process. Rather, it is more an optimization trick and bearing a default value of 0 will not cause any issues, since feature selection is done during classification.

3.1 Feature Extraction

As shown in Algorithm 1, feature extraction in MVGs follow the same paradigm as extracting features from individual graphs in an MVG and concatenating all features together, since graph features extracted in this paper are solely statistical and do not

pertain orders. Consequently, these features can be fed into any generic classification algorithms [13] well studied in the machine learning and data mining community. It is utterly important that this feature extraction transforms the sequential characteristics of time series data into unordered feature vectors, so that any modern classification algorithm can be taken advantage of.

Algorithm 1 Building time series MVGs and extracting features from them.

```

1: procedure EXTRACTFEATURES( $T$ )
2:    $\mathbb{F} \leftarrow \emptyset$  ▷ Feature set
3:    $\mathbb{G} \leftarrow \emptyset$  ▷ MVGs
4:    $T' \leftarrow T$ 
5:   while  $|T'| > \tau$  do ▷ Ignore trivial graphs
6:      $\mathbb{G} \leftarrow \mathbb{G} \cup \text{BuildVGAndHVG}(T')$ 
7:      $T' \leftarrow \text{DimensionalityReduction}(T)$  ▷  $|T'| \leq \frac{|T|}{2}$ 
8:   for  $G \in \mathbb{G}$  do
9:      $\mathbb{M} \leftarrow \text{MotifProbabilityDistribution}(G)$ 
10:     $\mathbb{M} \leftarrow \text{Normalize}(\mathbb{M})$ 
11:     $\mathbb{F} \leftarrow \mathbb{F} \cup \{\mathbb{M}, \text{OtherStatistics}(G)\}$ 
12:   return  $\mathbb{F}$ 

```

Although features have become unordered, it is nevertheless important to carefully curate them in order to achieve better classification results. Note that PGD is very efficient in counting motifs in graphs, and the dominant features from time series MVGs will be the probability distribution of motifs of different sizes:

Definition 3.4 (Motif probability distribution). Given a time series visibility graph G and the set of motifs \mathbb{M} , the motif probability distribution \mathbb{P}_G is the set of probabilities corresponding to each motif in \mathbb{M} .

Empirically, the distribution of connected and disconnected motifs vary greatly in graphs. It is thus desirable to calculate separately motif probability distributions depending upon motif connectivity. To that end, the motif probability distributions (MPDs) are calculated per motif size and connectivity. This essentially normalizes extracted motif features. Specifically, MPDs are normalized according to the following five groups (cf. Table 1): $\{\{M_{2_1}, M_{2_2}\}, \{M_{3_1}, M_{3_2}\}, \{M_{3_3}, M_{3_4}\}, \{M_{4_1}, \dots, M_{4_6}\}, \{M_{4_7}, \dots, M_{4_{11}}\}\}$. Finally, it is obvious that other graph features – e.g., graph density and assortativity coefficient – are independent of MPDs. As a result, no further curation for such features is needed.

3.2 Classification

When features are extracted, we can feed them into a generic classifier to learn from labeled samples and predict the class of unlabeled ones. We are in favor of most well-known and widely accepted and well optimized algorithms such as SVM, Random Forest and eXtreme Gradient Boosting (XGBoost) [8] for this task. Especially, as a highly optimized distributed gradient boosting library, XGBoost runs on major distributed environment. It has gained remarkable adoption since its inception and is well known for its performance in machine learning and data mining competitions.

Typical classification tasks involve learning models and selecting a performant estimator through the process of validation. Although in this study we have proposed a parameter-free method for feature extraction, it is still required to tune the hyper-parameters for generic classifiers. Note that hyper-parameters in machine learning refers to parameters that are external to the model and such values cannot be estimated from the training

data. As a result, they are often set using heuristics. For instance, the k in k NN classifier can be considered as a hyper-parameter, since “there is no analytical formula available to calculate an appropriate value” [22]. In order to tune hyper-parameters in our classifiers, we conduct cross-validation to evaluate the performance of estimators and apply grid search to find the most satisfactory estimator based on the cross entropy scores:

$$-\log P(\hat{y}|y) = -(y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})) \quad (5)$$

where y is the ground truth, and \hat{y} is the probabilistic predictions by an estimator. Since datasets may be highly imbalanced, which will lead to degraded classification results, we can apply random oversampling techniques over the minority class and use stratified cross validation to preserve class balance when validating models.

4 EVALUATION

To evaluate our approach, we conduct experiments using a large number of publicly accessible datasets. We first introduce the datasets. Then we list and validate our heuristics, followed by creating an accurate classifier with stacked generalization. Next, we compare the accuracy and efficiency of our method with state-of-the-art approaches. Finally, we close this section with a case study and discussions.

4.1 Datasets

When validating our heuristics, we consider the most popular and largest open dataset for TSC: the UCR Time Series Classification Archive [9]. Specifically, we use a subset from the UCR archive that are more recent (added after the summer of 2015), which include datasets from various fields ranging from electrocardiograms (ECG) to intra-species image recognition data. These datasets have a uniform file format and consistent internal data structures, making it possible to conduct batch processing in a content-agnostic manner. Furthermore, datasets in this subset are generally larger in size, making it more reliable to evaluate the scalability of TSC algorithms.

When comparing our approach with state-of-the-art approaches, we take advantage of the UEA & UCR Time Series Classification Repository since it contains all datasets from the UCR archive. In addition, this repository also provides open source implementation for more than 18 TSC algorithms [3] and benchmarking results are publicly available from the repository website¹. We compare our results with relevant algorithms using the default train and test split from this repository.

4.2 Validating Heuristics

Before diving directly into MVG representations, it would be a prerequisite to validate our heuristics step by step. We summarize these heuristics below:

- (1) Motif statistics from VGs and HVGs can serve better during classification when combined with other graph features such as degree statistics.
- (2) Features from HVGs are more capable of capturing *local* characteristics while those from VGs are capable of capturing *global* characteristics. Combining features from both HVGs and VGs can help yield more accurate classification results.

¹<http://timeseriesclassification.com/>

- (3) Multiscale representations are able to reveal time series features at different scales, and a generic classifier is able to conduct feature selection and find important features to perform more accurate classification than using features without multiscale augmentation.

When validating these heuristics, we feed the features corresponding to each heuristic to an XGBoost classifier for 3-fold cross-validation and model selection. A set of hyper-parameters have been set for grid search, including the learning rate (three choices from 0.01 to 0.3), number of estimators (10 choices from 10 to 100), max tree depth (10 or 20). In order to prevent overfitting, the subsampling and column sampling hyper-parameters are both set to 0.5 to randomly collect half of the data instances and features to grow trees. To reduce the impact of random over sampling on minority classes and floating-point summation issues in parallel processing, all experiments have been repeated five times and average accuracy scores are calculated. Finally, all experiments are conducted on a Linux server with two Intel Xeon E5-2430 CPUs with a clock rate of 2.20GHz. With this setup, we illustrate step by step how experiments are setup and present the final classification results in Table 2.

4.2.1 Choosing Graph Features. We first investigate which graph features are helpful for extracting distinguishable information from time series. Specifically, we transform time series into UVGs consisting of both VG and HVG representations and try to extract MPDs as well as other features (density, assortativity and degree statistics) from these graphs. We are interested in finding out whether MPDs are sufficient for TSC and if extracting other statistic features from graphs are necessary. Next, we take advantage of XGBoost classifier to train on different feature sets and classify the test datasets. Columns A, B, C and D in Table 2 presents the classification error rates using HVG with only MPDs, HVG with MPDs and other graph features, VG with only MPDs and VG with all features. Comparison on the bottom of the table shows that including features such as density and assortativity coefficient indeed helps improving classification accuracy. For HVGs, including features other than MPDs increases classification accuracy in 32 datasets, while for VGs 29 datasets saw accuracy improvements. A Wilcoxon signed rank test with p -values $9.48e-7$ and $9.56e-5$ suggests that such improvement is indeed significant (both p -values < 0.05). Figure 3 shows the classification results in the form of scatter plots, where each point represent a dataset. Such results along with the Wilcoxon test indeed suggest that our Heuristic (1) is valid.

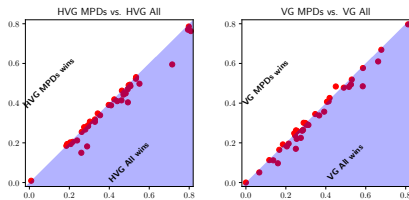


Figure 3: Comparison of classification error rates: using MPDs with or without other graph features.

4.2.2 VG and HVG. Next, it is necessary to show that graph features extracted from both VGs and HVGs are important. Recall that, intuitively, VGs are helpful for capturing global features while HVG can help locating local features. We separately test the distinguishing power of VGs and HVGs for time series in order to make sure that both can lead to satisfactory classification results.

Furthermore, we conduct experiments to investigate if combining VGs and HVGs can better capture both global and local features for time series and thus lead to more accurate classification.

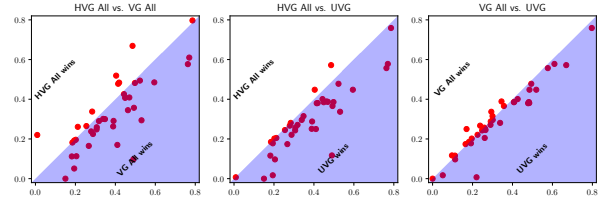


Figure 4: Comparison of classification error rates: using HVGs, VGs or combining two together (denoted as UVG here).

Figure 4 illustrates the comparison of classification accuracy with VG and HVG features as well as combining both VG and HVG. The first scatter plot shows that for majority datasets, VG features can yield more accurate classification performance. However, it is still worth mentioning that HVG features can also outperform VG features in some specific datasets, possibly due to the reason that local features are more influential in such datasets. Moreover, it is obvious from the other two scatter plots in Figure 4 that combining both VG features and HVG features can greatly boost the classification accuracy. This is probably due to the excellent feature selection capability of XGBoost, so that the classifier is able to find out which features are more important during the training process. In addition, as shown in Table 2, using VGs outperforms HVGs in 30 datasets with a p -value of $3.09e-3$, suggesting VGs are capable of capturing more characteristics in time series. Finally, combining features from both VGs and HVGs yields more accurate results in 27 datasets with a p -value of $5.01e-3$, indicating significant improvement when combining two different types of graphs. As a result, the validity of our Heuristic (2) can be confirmed.

4.2.3 UVG, AMVG and MVG. Since we have demonstrated that taking advantage of both VG and HVG features can improve classification accuracy for UVG representations, now we can investigate whether Heuristic (3) holds, *i.e.*, if multiscale representations can help achieving even more accurate results. To visually inspect which representation suites best for TSC, we further draw scatter plots of the accuracy results in Figure 5. It is then obvious that AMVG and UVG (scatter plot on top) lead to similar classification accuracy, which suggests that AMVG can be good approximations for original time series data. Furthermore, the two scatter plots in the bottom of Figure 5 indeed confirm that MVG representations result in better classification performance, since almost all dots representing results from different datasets fall on the side of MVG.

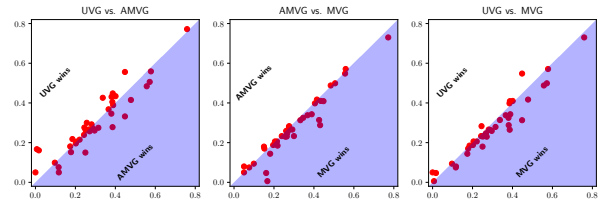


Figure 5: Comparison of UVG, AMVG and MVG's error rates.

From Table 2, we can see that multiscale approximations outperforms uniscale time series in 19 datasets, with a Wilcoxon p -value of $0.8623 > 0.05$. As a result, AMVG representations are not statistically significantly different than UVGs. On the other

Table 2: Error rates of classifying 39 UCR datasets compared with 1NN-Euclidean and 1NN-DTW. Different heuristic combinations are taken into account. Bold-faced values indicate lowest error rates (including ties) for specific datasets in all experiments.

Dataset	#Cls.	#Train	#Test	Dim.	Scales →		UVG				AMVG	MVG	
					Type of Graphs →		HVG		VG		VG+HVG		
					Features →		MPDs	All	MPDs	All	All		
					1NN-ED	1NN-DTW	A	B	C	D	E	F	G
ArrowHead	3	36	175	251	0.200	0.200	0.482	0.449	0.406	0.407	0.385	0.405	0.398
BeetleFly	2	20	20	512	0.250	0.300	0.440	0.410	0.250	0.170	0.250	0.150	0.180
BirdChicken	2	20	20	512	0.450	0.300	0.260	0.150	0.000	0.000	0.000	0.050	0.050
Computers	2	250	250	720	0.424	0.380	0.294	0.284	0.367	0.338	0.281	0.292	0.266
DistalPhalanxOutlineAgeGroup	3	139	400	80	0.218	0.228	0.204	0.202	0.214	0.196	0.202	0.196	0.188
DistalPhalanxOutlineCorrect	2	276	600	80	0.248	0.232	0.409	0.389	0.251	0.263	0.251	0.264	0.231
DistalPhalanxTW	6	139	400	80	0.273	0.272	0.342	0.348	0.298	0.300	0.315	0.275	0.279
ECG5000	5	500	4500	140	0.075	0.075	0.289	0.182	0.137	0.112	0.116	0.076	0.075
Earthquakes	2	139	322	512	0.326	0.258	0.262	0.255	0.286	0.265	0.245	0.276	0.283
ElectricDevices	7	8926	7711	96	0.450	0.376	0.503	0.493	0.392	0.357	0.366	0.368	0.338
FordA	2	1320	3601	500	0.341	0.341	0.009	0.009	0.254	0.220	0.007	0.167	0.006
FordB	2	810	3636	500	0.442	0.414	0.328	0.318	0.313	0.290	0.271	0.257	0.230
Ham	2	109	105	431	0.400	0.400	0.463	0.463	0.347	0.345	0.389	0.389	0.343
HandOutlines	2	370	1000	2709	0.199	0.197	0.293	0.288	0.275	0.225	0.221	0.215	0.206
Herring	2	64	64	512	0.484	0.469	0.425	0.419	0.450	0.484	0.381	0.431	0.288
InsectWingbeatSound	11	220	1980	256	0.438	0.422	0.808	0.763	0.586	0.577	0.557	0.484	0.488
LargeKitchenAppliances	3	375	375	720	0.507	0.205	0.461	0.414	0.490	0.478	0.380	0.346	0.325
Meat	3	60	60	448	0.067	0.067	0.497	0.490	0.160	0.097	0.117	0.050	0.080
MiddlePhalanxOutlineAgeGroup	3	154	400	80	0.260	0.253	0.274	0.279	0.246	0.238	0.267	0.266	0.247
MiddlePhalanxOutlineCorrect	2	291	600	80	0.247	0.318	0.534	0.531	0.305	0.294	0.337	0.426	0.314
MiddlePhalanxTW	6	154	399	80	0.439	0.419	0.471	0.443	0.419	0.426	0.401	0.434	0.410
PhalangesOutlinesCorrect	2	1800	858	80	0.239	0.239	0.397	0.391	0.302	0.291	0.288	0.272	0.264
Phoneme	39	214	1896	1024	0.891	0.773	0.798	0.786	0.812	0.797	0.759	0.772	0.730
ProximalPhalanxOutlineAgeGroup	3	400	205	80	0.215	0.215	0.207	0.194	0.185	0.192	0.178	0.152	0.170
ProximalPhalanxOutlineCorrect	2	600	291	80	0.192	0.210	0.280	0.267	0.167	0.165	0.174	0.181	0.144
ProximalPhalanxTW	6	205	400	80	0.292	0.263	0.303	0.307	0.257	0.259	0.257	0.300	0.233
RefrigerationDevices	3	375	375	720	0.605	0.560	0.533	0.523	0.526	0.494	0.478	0.415	0.417
ScreenType	3	375	375	720	0.640	0.589	0.506	0.486	0.678	0.669	0.572	0.506	0.499
ShapeletSim	2	20	180	500	0.461	0.300	0.189	0.194	0.067	0.051	0.017	0.161	0.047
ShapesAll	60	600	600	512	0.248	0.198	0.715	0.595	0.585	0.485	0.448	0.332	0.313
SmallKitchenAppliances	3	375	375	720	0.659	0.328	0.239	0.212	0.282	0.261	0.205	0.205	0.206
Strawberry	2	370	613	235	0.062	0.062	0.217	0.205	0.117	0.113	0.097	0.099	0.094
ToeSegmentation1	2	40	228	277	0.320	0.250	0.354	0.339	0.289	0.301	0.296	0.261	0.259
ToeSegmentation2	2	36	130	343	0.192	0.092	0.185	0.185	0.205	0.182	0.185	0.218	0.185
UWaveGestureLibraryAll	8	896	3582	945	0.052	0.034	0.551	0.498	0.516	0.482	0.386	0.278	0.265
Wine	2	57	54	234	0.389	0.389	0.493	0.404	0.530	0.519	0.448	0.556	0.548
WordSynonyms	25	267	638	270	0.382	0.252	0.794	0.770	0.662	0.610	0.578	0.559	0.571
Worms	5	77	181	900	0.635	0.586	0.492	0.470	0.414	0.409	0.387	0.448	0.409
WormsTwoClass	2	77	181	900	0.414	0.414	0.328	0.306	0.242	0.248	0.243	0.239	0.233
Comparison versus					G	G	B	D	D	E	F	G	E
Number of more accurate datasets					26	23	32	30	29	27	19	29	30
Wilcoxon test p -value					0.01	0.1638	9.48e-7	3.09e-3	9.56e-5	5.01e-3	0.8623	1.72e-4	8.74e-4

hand, MVGs outperform AMVG in 29 datasets with a p -value of $1.72e-4$ and MVGs are more accurate than UVGs in 30 datasets with a p -value of $8.74e-4$, suggesting that MVG representations indeed contribute significantly towards a more accurate feature extraction and TSC process. As a result, our Heuristic (3) is valid.

4.2.4 Summary of Heuristic Validation. Overall, all our heuristics are supported by experiment results and each heuristic contributes significantly to a more accurate TSC process. Table 2 also shows the comparison between our approach and Euclidean distance- as well as DTW-based nearest neighbor classification. Our approach outperforms 1NN-Euclidean in 26 datasets with a Wilcoxon p -value of 0.01 , suggesting MVG features with XGBoost is significantly more accurate than 1NN-Euclidean. Moreover, MVG appears to be in par with 1NN-DTW in terms of classification accuracy since 23 datasets are in favor of MVG with a p -value of 0.1638 . Next, we will try to build a more robust and accurate classifier using stacked generalization.

4.3 Stacked Generalization

Previously we have solely taken advantage of XGBoost for classifying time series with features extracted from graphs. However, it can also be interesting to investigate if other classifiers can achieve similar results. Besides, although modern classifiers such as XGBoost and RF are capable of efficiently conducting feature selection during training, we can still conduct feature selection processes and feed such *a priori* information to classifiers in order to achieve better classification accuracy. To that end, we propose feeding different sets of features to a collection of classifiers and then create a meta-classifier using stacked generalization (*a.k.a* stacking or blending) [41]. This meta-classifier will then hopefully generate better final predictions. In fact, a variant of this technique has led to winning the Netflix Prize with a reward of one million dollars [37].

Before building a meta-classifier with stacked generalization, we first make sure that features we previously extracted are suitable inputs for different classifiers such as RF and SVM. Generally, tree-based classifiers are not so sensitive to monotonic transformations of individual features. As a result, it is often not required

to have features scaled to similar magnitudes for RF and XGBoost. However, since SVM’s kernel functions (in an Euclidean space) are usually sensitive to different feature magnitudes, we use Min-Max scaling to transform each feature into range of zero and one. After that, we compare the classification performance of these three classifiers.

In order to evaluate the significance of the differences a generic classifier can incur, we take advantage of the Nemenyi test [12], which is a post-hoc test aiming for finding whether groups of data differ after a statistical test of multiple comparisons. This test can be illustrated by means of a critical difference diagram, where average ranks of all approaches are presented. Specifically, the location of vertical lines indicate the average ranking of an approach and bold lines (insignificance lines) indicate groups of approaches that are not significantly different. In this case, for one approach to be considered significantly better than another, its overall ranking has to be at least 0.5307 higher than its competitor. As shown in Figure 6, XGBoost performs slightly better than RF in general, and both are significantly more accurate than SVM. Such results are not surprising, since [13] have empirically tested hundreds of classifiers and concluded that RF produces the most accurate classification results. This study was conducted before the initial release of XGBoost, and the recent adoption momentum of XGBoost indeed suggests that XGBoost is great for yielding accurate classification results.

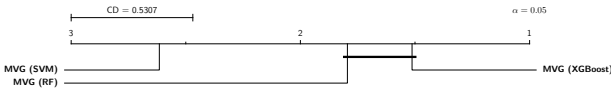


Figure 6: Critical difference diagram comparison of RF, SVM and XGBoost.

Now that generic classifiers can be used for graph features extracted from time series, we then set to investigate whether stacking can help further increase classification accuracy. We first stack top performing classifiers in each family before blending classifiers from different families. Specifically, we first select the top five most accurate classifiers from RF, SVM and XGBoost through cross validation. Then these five classifiers are stacked to produce a meta-classifier, which is used for producing final predictions. Finally, when stacking classifiers of different families, five classifiers from each family have been selected, thus the meta-classifier has been trained with 15 classifiers. Our algorithm is described in Algorithm 2.

Algorithm 2 Algorithm for creating an ensemble classifiers using stacked generalization.

Input: Training dataset $\mathbb{D} = \{\mathbb{X}_i, y_i\}_{i=1}^m$ ($\mathbb{X}_i \in \mathbb{R}^n$, $y_i \in \mathbb{Z}$)
Base classifiers \mathbb{H} (with different hyper-parameters)
Output: An stacked ensemble E

```

1: procedure BUILDSTACKINGENSEMBLE( $\mathbb{D}$ ,  $\mathbb{H}$ )
2:    $\mathbb{S} \leftarrow \text{CreateStratifiedKFolds}(\mathbb{D}, \text{cv}=3)$  ▷ 3-fold CV
3:    $\mathbb{E} \leftarrow \emptyset$  ▷ Best performing base estimators
4:   for all  $h \in \mathbb{H}$  do
5:      $H \leftarrow \emptyset$ 
6:     for all  $S = \{\mathbb{D}_{train}, \mathbb{D}_{validation}\} \in \mathbb{S}$  do
7:       TrainClassifier( $h, \mathbb{D}_{train}$ )
8:        $\hat{y} \leftarrow \text{Predict}(h, \mathbb{X}_{validation})$ 
9:       score  $\leftarrow -\log P(\hat{y}|y_{validation})$ 
10:       $H \leftarrow H \cup \{h, \text{score}\}$ 
11:    $H \leftarrow \text{arg sort}(H)$  ▷ Sort estimators by score
12:    $\mathbb{E} \leftarrow \mathbb{E} \cup \text{slice}(H, k)$  ▷ Select top- $k$  estimators
13:    $W \leftarrow \text{ComputeEstimatorWeights}(\mathbb{E})$  ▷ with logistic regression
14:    $E \leftarrow \sum_{i=1}^{|\mathbb{E}|} W_i \mathbb{E}_i$ 
15:   return  $E$ 

```

Our experiments show that, for RF and SVM, stacking their top performing classifiers indeed increases final classification. In the case of XGBoost, however, stacking its most accurate classifiers does not seem to significantly increase classification accuracy, since the classification results of stacked generalization is on par with those with a single most accurate classification during cross validation. It possibly indicates that XGBoost has already very good generalization capabilities. Finally, stacking most accurate classifiers from three different families can help achieving better classification accuracy than single best XGBoost classifier in most datasets. As a result, stacked generalization can indeed be helpful for further improving the performance of MVG.

Figure 7 demonstrates how stacked generalization can help boosting classification accuracy. It is straightforward that stacking XGBoost and SVM produces similar classification accuracy, while stacking top performers from all three families can be significantly more accurate than using a single family. As a result, we are confident that staked generalization is favorable for more accurate TSC.

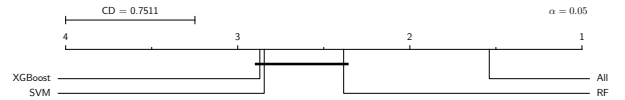


Figure 7: Critical difference diagram comparison of stacking single family of classifiers versus all families of classifiers.

4.4 Accuracy Benchmarking

Finally, we compare our results with the state-of-the-art and relevant approaches. Table 3 presents the classification error rates as well as the running time statistics. Specifically, datasets in this experiment are from the UEA & UCR Time Series Classification Repository thanks to the many benchmarking results that are publicly available. Note that although names of the datasets used in this repository are exactly the same compared to our previous experiments, the similarity of their contents is not guaranteed. In fact, the training and testing datasets may have been swapped for a number of datasets. An obvious example is the FordA dataset, where in this experiment the training dataset and testing set are of size 1320 and 3601 respectively, while in our previous experiments the training set has 3601 samples and the testing test has 1320.

We compare our method MVG with five state-of-the-art approaches: two distance-based global similarity matching algorithms including Euclidean- and DTW-based nearest neighbor classification (1NN-ED and 1NN-DTW), and three local pattern matching algorithms including Learning Shapelets (LS) [15], Fast Shapelets (FS) [31] and SAX-VSM [36]. Among all five approaches, LS is recognized as the most accurate classifier [39] by the research community. However, LS is also known for its high computation complexity. We first compare the classification accuracy of different approaches in this section and then investigate MVG’s efficiency later in section 4.5.

Viewing the error rate comparison columns in Table 3, it is obvious that MVG is the most accurate classifier with 16 winning datasets. LS then follows MVG with 12 winning datasets. A Wilcoxon test between MVG and LS yields a p -value of $0.3421 > 0.05$, suggesting that MVG should not be considered significantly better than LS. SAX-VSM is ranked third with 10 winning cases. The Wilcoxon test again suggest that such difference is not statistically significant. However, MVG is indeed significantly more

Table 3: Classification error rates compared with five benchmark approaches and running time statistics (in seconds).

Dataset	# Cls.	#Train	#Test	Dim.	Classification Error Rate						MVG Runtime			FS	
					1NN-ED	1NN-DTW	LS	FS	SAX-VSM	MVG	FE	Clf.	Σ	Runtime	
ArrowHead	3	36	175	251	0.200	0.297	0.154	0.406	0.211	0.371	8	29	37	30	
BeetleFly	2	20	20	512	0.250	0.300	0.200	0.300	0.100	0.050	2	21	23	54	
BirdChicken	2	20	20	512	0.450	0.250	0.200	0.250	0.000	0.000	4	22	26	38	
Computers	2	250	250	720	0.424	0.300	0.416	0.500	0.380	0.252	22	51	73	1293	
DistalPhalanxOutlineAgeGroup	3	400	139	80	0.374	0.230	0.281	0.345	0.158	0.254	11	86	97	45	
DistalPhalanxOutlineCorrect	2	600	276	80	0.283	0.283	0.221	0.250	0.272	0.281	19	93	112	101	
DistalPhalanxTW	6	400	139	80	0.367	0.410	0.374	0.374	0.396	0.381	12	204	216	59	
ECG5000	5	500	4500	140	0.075	0.076	0.068	0.077	0.090	0.069	162	190	352	170	
Earthquakes	2	322	139	512	0.288	0.281	0.259	0.295	0.252	0.252	22	78	100	3689	
ElectricDevices	7	8926	7711	96	0.448	0.398	0.413	0.421	0.295	0.332	406	6344	6750	3558	
FordA	2	3601	1320	500	0.335	0.445	0.043	0.213	0.173	0.014	207	410	617	44832	
FordB	2	3636	810	500	0.394	0.380	0.083	0.272	0.249	0.333	183	534	717	45874	
Ham	2	109	105	431	0.400	0.533	0.333	0.352	0.190	0.343	8	32	40	670	
HandOutlines	2	1000	370	2709	0.138	0.119	0.519	0.189	0.092	0.200	4431	182	4613	179745	
Herring	2	64	64	512	0.484	0.469	0.375	0.469	0.375	0.344	19	30	49	286	
InsectWingbeatSound	11	220	1980	256	0.438	0.645	0.394	0.511	0.453	0.459	85	130	215	705	
LargeKitchenAppliances	3	375	375	720	0.507	0.205	0.299	0.440	0.123	0.288	32	89	121	7301	
Meat	3	60	60	448	0.150	0.067	0.100	0.067	0.067	0.050	10	31	41	120	
MiddlePhalanxOutlineAgeGroup	3	400	154	80	0.481	0.500	0.429	0.455	0.455	0.435	9	88	97	50	
MiddlePhalanxOutlineCorrect	2	600	291	80	0.234	0.302	0.220	0.271	0.323	0.289	15	87	102	80	
MiddlePhalanxTW	6	399	154	80	0.487	0.494	0.494	0.468	0.513	0.481	9	177	186	62	
PhalangesOutlinesCorrect	2	1800	858	80	0.239	0.272	0.235	0.256	0.290	0.248	48	209	257	332	
Phoneme	39	214	1896	1024	0.891	0.772	0.782	0.826	0.895	0.692	134	1419	1553	25604	
ProximalPhalanxOutlineAgeGroup	3	400	205	80	0.215	0.195	0.166	0.220	0.176	0.166	11	70	81	41	
ProximalPhalanxOutlineCorrect	2	600	291	80	0.192	0.216	0.151	0.196	0.172	0.144	18	80	98	80	
ProximalPhalanxTW	6	400	205	80	0.293	0.239	0.224	0.298	0.390	0.220	12	160	172	49	
RefrigerationDevices	3	375	375	720	0.605	0.536	0.485	0.667	0.347	0.421	37	77	114	12798	
ScreenType	3	375	375	720	0.640	0.603	0.571	0.587	0.488	0.480	35	89	124	8473	
ShapeletSim	2	20	180	500	0.461	0.350	0.050	0.000	0.283	0.000	6	22	28	76	
ShapesAll	60	600	600	512	0.248	0.232	0.232	0.420	0.302	0.255	168	1833	2001	7939	
SmallKitchenAppliances	3	375	375	720	0.656	0.357	0.336	0.667	0.421	0.208	30	75	105	5667	
Strawberry	2	613	370	235	0.054	0.059	0.089	0.097	0.043	0.076	29	75	104	314	
ToeSegmentation1	2	40	228	277	0.320	0.228	0.066	0.044	0.070	0.197	8	26	34	30	
ToeSegmentation2	2	36	130	343	0.192	0.162	0.085	0.308	0.138	0.185	6	22	28	38	
UWaveGestureLibraryAll	8	896	3582	945	0.052	0.108	0.047	0.211	0.201	0.258	470	343	813	32047	
Wine	2	57	54	234	0.389	0.426	0.500	0.241	0.037	0.519	4	27	31	22	
WordSynonyms	25	267	638	270	0.382	0.351	0.393	0.569	0.509	0.522	39	1017	1056	907	
Worms	5	181	77	900	0.545	0.416	0.390	0.351	0.442	0.182	26	98	124	6852	
WormsTwoClass	2	181	77	900	0.390	0.377	0.273	0.273	0.286	0.130	27	45	72	5044	
Number of best (including ties)					1	2	12	3	10	16				24	15
Wilcoxon test p -value					0.0023	0.0044	0.3421	0.0005	0.5767	-	Total time \rightarrow			21379	395075

accurate than FS, 1NN-DTW and 1NN-ED. Moreover, scatter plots in Figure 8 shows that most of the points in each comparison are located away from the diagonal line, suggesting that MVG is indeed a very different approach when compared to the state-of-the-art. Having confirmed MVG’s excellent accuracy, we continue to investigate its runtime efficiency.

4.5 Efficiency

The pipeline of MVG consists of a feature extraction phase and a train-validate-test process. During the feature extraction process, time series are firstly transformed in to VGs and then features are extracted from such graphs. With optimization from [1], transforming time series of length n into VGs has a computation complexity of $O(n \log^2(n))$ and it can be effectively solved within $O(\log^2(n))$ time when taking full advantage of parallelization. Extracting motif features from VGs may be potentially expensive, fortunately PGD provides a fully parallel way of counting small motifs. For instance, counting graph cliques of size 4 – one of the most time consuming tasks in PGD – has a computation complexity of $O(m \cdot \Delta \cdot T_{max})$, where m is the number of 4-motifs (i.e., 11), Δ is the maximum degree and T_{max} is the maximum number of triangles incident to an edge and $T_{max} \ll \Delta$. PGD is hundreds of times faster than other motif counting algorithms:

counting motifs from a graph with more than 26,000 vertices can take only 0.01 seconds on a twelve-core commodity CPU [2]. Since other statistic features such as density, k -core, assortativity and degree statistics extracted from time series VGs are intentionally chosen to be simple metrics, collecting these features has a time complexity of $O(\max(|V|, |E|))$. Since we use a multiscale graph representation, ultimately graph generation has a mono-thread complexity of $O(n \log^3(n))$ and in parallel an $O(\log^3(n))$ time complexity. As a result, feature extraction per graph has a computation complexity of $O(\max(|V|, |E|) + m \cdot \Delta \cdot T_{max})$. The classification process leverages state-of-the-art classifiers that are widely used and well-optimized. Overall, the efficiency of MVG may be best illustrated when we compare our method against our benchmarking approaches.

Previous research [39] has shown that LS is extremely time consuming, and FS as an approximated approach can be 100X faster than LS. As a result, FS will be a good and strong baseline to which the running time of our approach can be compared. Thus we record the running time of FS and MVG and compare their efficiency. These experiments are conducted on a computer with Intel i7-4980HQ quad-core CPU clocked at 2.80GHz, 16GB memory and solid-state drive suitable for fast I/O. We use the

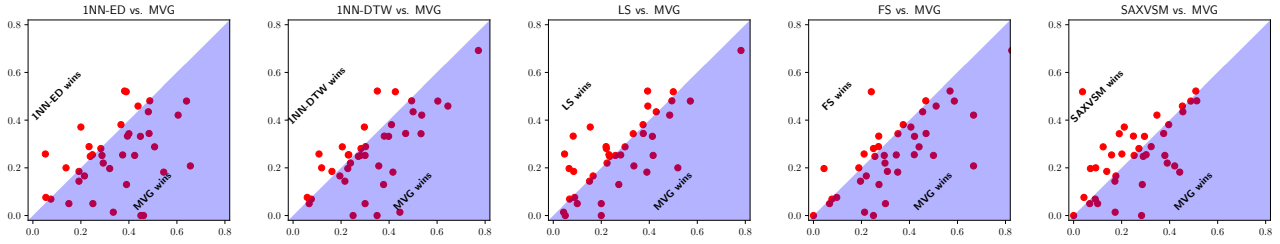


Figure 8: Comparison of classification accuracy with five state-of-the-art approaches in the form of scatter plots.

FS implementation by its original authors [31] with default parameters. Columns located on the right side of Table 3 shows the running time per dataset for MVG and FS. For MVG, running time for feature extraction and training are recorded separately and then summed. Overall, MVG completes faster than FS for 24 datasets. The total running time of FS for all 39 datasets amounts to 395075 seconds, which is more than 18 times that of MVG. A scatter plot of the running time is provided in Figure 9, obviously MVG can be up to 100X faster than FS, suggesting that it is indeed an efficient TSC approach. In addition, we note that FS appears to be time consuming with large datasets with time series of high dimensionality, while the running time of MVG remains reasonable as datasets grow larger. Furthermore, since we have experimented only on a quad-core commodity computer that is not really powerful in terms of parallel processing, the efficiency of MVG can be easily boosted by adding more computing cores. Overall, MVG can indeed be considered an efficient TSC approach.

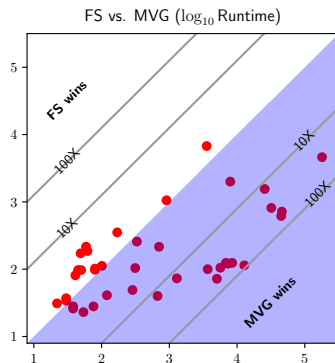


Figure 9: Runtime comparison between FS and MVG.

4.6 Case Study

Although accuracy and efficiency are very important measurements for TSC approaches, it is also desirable for TSC methods to have comprehensible classification processes, so that users may gain extra insights into his/her time series data. Popular approaches based on shapelets and subsequences may have a natural advantage in classification comprehensibility. However, since features extracted in MVG are solely statistical, it can be difficult to locate exactly where a distinguishing subsequence lies in the original time series. However, for MVG it is still possible to understand which features have contributed most to correct classification. For instance, when feeding features to train an XGBoost classifier, it will assign weights to all the features. After ranking these features, we can also gain insights into the data. For instance, Figure 10 illustrates a scatter matrix plot for the test dataset of FordA showing ten most important features

learned by the classifier². Out of ten features, six are features from HVGs, which are created from the original series (T_0). VGs features from dimensionality-reduced approximations (T_2 and T_3) are also present. Moreover, it appears that most highly ranked graph features for this dataset are MPDs and assortativity, suggesting that statistics other than MPDs are indeed helpful. Finally, visually from the kernel density estimation it is obvious that some features alone – e.g., T_0 HVG $P(M4_4)$ – can already provide good classification guidelines, suggesting that it is feasible for MVG to present visually comprehensible cues regarding its classification decisions.

4.7 Discussions

Due to specific characteristics of VGs, obviously MVG may not be suitable for every TSC scenarios. For instance, VGs are agnostic of affine transformations in time series. That is, in applications where the absolute oscillation is more important, MVG is less likely to detect such characteristics. Furthermore, when dealing with non-stationary time series data where trends are very frequently found, MVG works best when trends are not the deciding factor since VGs may not be able to capture long-term monolithic trends. Since many graph features have been taken into account in MVG, we believe it can be robust in practical applications.

In addition to limitations inherited from VGs, our approach is more suitable for applications where the size of training datasets is larger, so that classification models can generalize better. In fact, when we review the classification accuracy of MVG, it seems that a majority of its winning datasets have either long time series or large training datasets. This is perhaps innate to the nature of statistics: sample size has to be sufficiently large to make accurate estimations. Based on running time comparison, MVG also scales well with large datasets, which can be an important advantage in the era of big data.

5 RELATED WORK

TSC is a major task in time series mining thanks to its wide application scenarios. As a consequence, there are a plethora of classification algorithms for TSC. Classical TSC approaches involve utilizing 1NN classification together with similarity measures specific to time series data, e.g., DTW [7] and its variants with lower bounding [33] and early abandoning techniques [30].

Another line of research transforms time series into texts and resort to text classification algorithms. For example, inspired by the well known *bag-of-words* approach, [27] proposes the Bag-of-Patterns approach for TSC. SAX-VSM [36] also takes advantage of bag-of-words approach and builds term frequency-inverse document frequency (*TF-IDF*) vectors in its training phase. It defines a similarity measure of two vectors (that are constructed

²To save space, we show more illustrations comparing original time series and important MVG features on project website. URL: <http://daoyuan.li/mvg/>

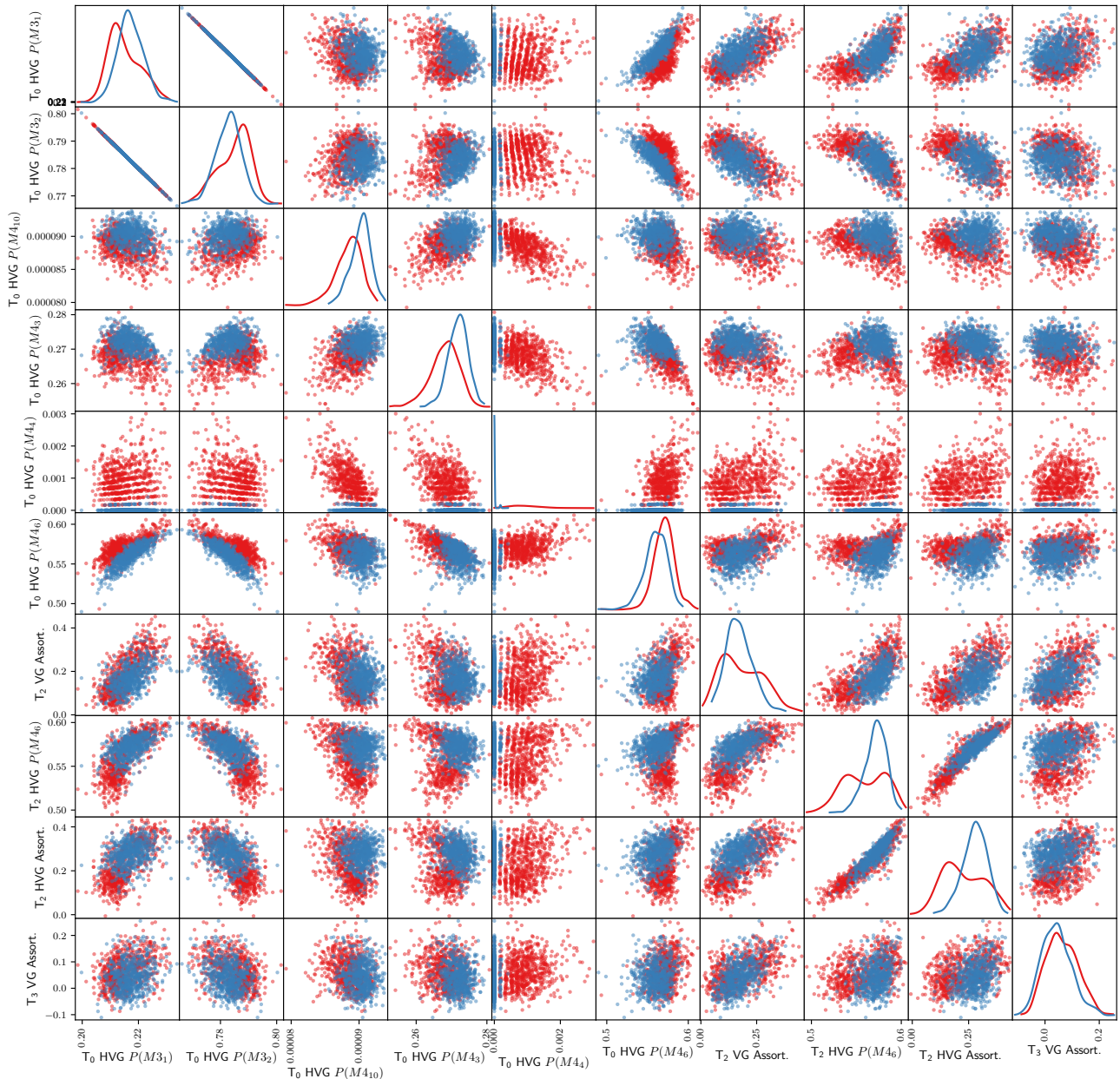


Figure 10: Scatter matrix of ten most important features for FordA's test dataset. Different point colors indicate different classes and the diagonal shows the Gaussian kernel density estimation for each feature.

from original series) based on their inner product. Both Bag-of-Patterns and SAX-VSM rely on SAX [26] for transforming time series into texts. A more recent work, Representative Pattern Mining (RPM) [39], also tries to classify time series by means of finding the most representative SAX-symbolized subsequences. Our previous work Domain Series Corpus (DSCo) [25] also transforms time series into texts and approximates TSC to a pseudo language detection problem. [35] invents another symbolic representation based on Fourier transform and proposes a bag-of-patterns approach named BOSS ensemble based on this symbolic representation method.

Recently, shapelet-based approaches are gaining popularity among the research community. A shapelet [44] is a single subsequence in time series that is representative of its class. However, these approaches [15, 16] are known for their high computation complexity. As we have demonstrated in this paper, FS [31] as

an approximated approach can also take long time to run. High computation complexity is also present for TSC approaches using deep neural networks [43]. Finally, [4] introduce an ensemble algorithm named Collective of Transformation-based Ensembles (COTE) – an ensemble of 35 different classifiers – that has shown to be very accurate but has a time complexity of $O(m^4 n^2)$, where m is the dimensionality of time series and n is the size of training dataset.

Although the concept of time series VGs has appeared for almost a decade, using it for TSC has just started picking up. Machine learning approaches taking advantage of VG and its variants are generally using artificially generated data [42] or EEG signals. Finally, graph kernel methods [21] can be used for evaluating graph similarity, which may potentially be used for TSC as well. Our approach transforms TSC into a graph classification [14] problem. Since we convert time series into

a set of graphs at different scales, we have not experimented generic graph classification algorithms. However, they are indeed interesting for future experiments.

6 CONCLUSIONS AND FUTURE WORK

This paper proposes a graph-based multiscale time series representation named MVG and a feature extraction method for TSC. MVG transforms time series into a collection of visibility graphs of various scales and feature extraction are conducted by investigating statistical graph features such as probability distributions of small motifs, assortativity as well as degree statistics. After a large scale evaluation with open datasets and comparison with a number of state-of-the-art TSC algorithms, our proposed approach appears to be both accurate and efficient: it can be more accurate than LS and up to $\sim 100X$ faster than FS.

In the future, we plan to further investigate other useful and efficient graph features – such as degree distribution entropy, centrality, bipartivity, *etc.* [11] – for MVG in order to further improve its accuracy. Currently we have only evaluated MVG with univariate time series data, we are also excited to investigate the possibility of adopting MVG for multivariate TSC.

ACKNOWLEDGMENT

The results of this study comes from an industrial collaboration project, the authors would like to thank Paul Wurth S.A. for supporting this project.

REFERENCES

- [1] Peyman Afshani, Mark de Berg, Henri Casanova, Benjamin Karsin, Colin Lambrechts, Nodari Sitchinava, and Constantinos Tsirogiannis. 2017. An Efficient Algorithm for the 1D Total Visibility-Index Problem. In *2017 Proceedings of the Nineteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 218–231.
- [2] Nesreen K. Ahmed, Jennifer Neville, Ryan A. Rossi, and Nick Duffield. 2015. Efficient Graphlet Counting for Large Networks. In *ICDM*. 1–10.
- [3] Anthony Bagnall, Jason Lines, Aaron Bostrom, James Large, and Eamonn Keogh. 2017. The great time series classification bake off: a review and experimental evaluation of recent algorithmic advances. *Data Mining and Knowledge Discovery* 31, 3 (2017), 606–660.
- [4] Anthony Bagnall, Jason Lines, Jon Hills, and Aaron Bostrom. 2015. Time-series classification with COTE: the collective of transformation-based ensembles. *IEEE Transactions on Knowledge and Data Engineering* 27, 9 (2015), 2522–2535.
- [5] Vladimir Batagelj and Matjaz Zaversnik. 2003. An $O(m)$ algorithm for cores decomposition of networks. *arXiv preprint cs/0310049* (2003).
- [6] Gustavo EAPA Batista, Xiaoyue Wang, and Eamonn Keogh. 2011. A Complexity-Invariant Distance Measure for Time Series.. In *SDM*, Vol. 11. 699–710.
- [7] Donald J Berndt and James Clifford. 1994. Using Dynamic Time Warping to Find Patterns in Time Series.. In *KDD workshop*, Vol. 10. 359–370.
- [8] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. ACM, 785–794.
- [9] Yanping Chen, Eamonn Keogh, Bing Hu, Nurjahan Begum, Anthony Bagnall, Abdullah Mueen, and Gustavo Batista. 2015. The UCR Time Series Classification Archive. (July 2015). www.cs.ucr.edu/~eamonn/time_series_data/.
- [10] Albert Cohen, Ingrid Daubechies, and Pierre Vial. 1993. Wavelets on the interval and fast wavelet transforms. *Applied and computational harmonic analysis* 1, 1 (1993), 54–81.
- [11] Luciano da F Costa, Francisco A Rodrigues, Gonzalo Travieso, and Paulino Ribeiro Villas Boas. 2007. Characterization of complex networks: A survey of measurements. *Advances in physics* 56, 1 (2007), 167–242.
- [12] Olive Jean Dunn. 1964. Multiple comparisons using rank sums. *Technometrics* 6, 3 (1964), 241–252.
- [13] Manuel Fernández-Delgado, Eva Cernadas, Senén Barro, and Dinani Amorim. 2014. Do we need hundreds of classifiers to solve real world classification problems. *J. Mach. Learn. Res* 15, 1 (2014), 3133–3181.
- [14] Thomas Gärtner, Peter Flach, and Stefan Wrobel. 2003. On graph kernels: Hardness results and efficient alternatives. *Learning Theory and Kernel Machines* (2003), 129–143.
- [15] Josif Grabocka, Nicolas Schilling, Martin Wistuba, and Lars Schmidt-Thieme. 2014. Learning time-series shapelets. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 392–401.
- [16] Jon Hills, Jason Lines, Edgaras Baranauskas, James Mapp, and Anthony Bagnall. 2014. Classification of time series by shapelet transformation. *Data Mining and Knowledge Discovery* 28, 4 (2014), 851–881.
- [17] Bing Hu, Yanping Chen, and Eamonn Keogh. 2013. Time series classification under more realistic assumptions. In *Proceedings of the 2013 SIAM International Conference on Data Mining*. SIAM, 578–586.
- [18] Jacopo Iacovacci and Lucas Lacasa. 2016. Sequential motif profile of natural visibility graphs. *Physical Review E* 94, 5 (2016), 052309.
- [19] Eamonn Keogh. 1997. Fast similarity search in the presence of longitudinal scaling in time series databases. In *Tools with Artificial Intelligence, 1997. Proceedings., Ninth IEEE International Conference on*. IEEE, 578–584.
- [20] Eamonn Keogh and Michael Pazzani. 2000. Scaling up dynamic time warping for datamining applications. In *Proceedings of the 6th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 285–289.
- [21] Risi Kondor and Horace Pan. 2016. The multiscale Laplacian graph kernel. In *Advances in Neural Information Processing Systems*. 2990–2998.
- [22] Max Kuhn and Kjell Johnson. 2013. *Applied predictive modeling*. Vol. 810. Springer.
- [23] Lucas Lacasa, Bartolo Luque, Fernando Ballesteros, Jordi Luque, and Juan Carlos Nuno. 2008. From time series to complex networks: The visibility graph. *Proceedings of the National Academy of Sciences* 105, 13 (2008), 4972–4975.
- [24] Lucas Lacasa, Bartolo Luque, Jordi Luque, and Juan Carlos Nuno. 2009. The visibility graph: A new method for estimating the Hurst exponent of fractional Brownian motion. *EPL (Europhysics Letters)* 86, 3 (2009), 30001.
- [25] Daoyuan Li, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. DSCo-NG: A Practical Language Modeling Approach for Time Series Classification. In *The 15th International Symposium on Intelligent Data Analysis*.
- [26] Jessica Lin, Eamonn Keogh, Li Wei, and Stefano Lonardi. 2007. Experiencing SAX: a novel symbolic representation of time series. *Data Mining and knowledge discovery* 15, 2 (2007), 107–144.
- [27] Jessica Lin, Rohan Khade, and Yuan Li. 2012. Rotation-invariant similarity in time series using bag-of-patterns representation. *Journal of Intelligent Information Systems* 39, 2 (2012), 287–315.
- [28] Bartolo Luque, Lucas Lacasa, Fernando Ballesteros, and Jordi Luque. 2009. Horizontal visibility graphs: Exact results for random time series. *Physical Review E* 80, 4 (2009), 046103.
- [29] Mark EJ Newman and Michelle Girvan. 2003. Mixing patterns and community structure in networks. *Statistical mechanics of complex networks* (2003), 66–87.
- [30] Thanawin Rakthanmanon, Bilson Campana, Abdullah Mueen, Gustavo Batista, Brandon Westover, Qiang Zhu, Jesin Zakaria, and Eamonn Keogh. 2012. Searching and mining trillions of time series subsequences under dynamic time warping. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 262–270.
- [31] Thanawin Rakthanmanon and Eamonn Keogh. 2013. Fast shapelets: A scalable algorithm for discovering time series shapelets. In *Proceedings of the thirteenth SIAM conference on data mining*.
- [32] Karl Johan Åström. 1969. On the choice of sampling rates in parametric identification of time series. *Information Sciences* 1, 3 (1969), 273–278.
- [33] Chotirat Ann Ratanamahatana and Eamonn Keogh. 2005. Three myths about dynamic time warping data mining. In *Proceedings of SIAM International Conference on Data Mining*. 506–510.
- [34] Pedro Ribeiro, Fernando Silva, and Luis Lopes. 2010. Efficient parallel sub-graph counting using g-tries. In *Cluster Computing (CLUSTER), 2010 IEEE International Conference on*. IEEE, 217–226.
- [35] Patrick Schäfer. 2015. The BOSS is concerned with time series classification in the presence of noise. *Data Mining and Knowledge Discovery* 29, 6 (2015), 1505–1530.
- [36] Pavel Senin and Sergey Malinchik. 2013. SAX-VSM: Interpretable time series classification using SAX and vector space model. In *IEEE 13th International Conference on Data Mining*. IEEE, 1175–1180.
- [37] Joseph Sill, Gábor Takács, Lester Mackey, and David Lin. 2009. Feature-weighted linear stacking. *arXiv preprint arXiv:0911.0460* (2009).
- [38] Supriya Supriya, Siuly Siuly, Hua Wang, Jinli Cao, and Yanchun Zhang. 2016. Weighted visibility graph with complex network features in the detection of epilepsy. *IEEE Access* 4 (2016), 6554–6566.
- [39] Xing Wang, Jessica Lin, Pavel Senin, Tim Oates, Sunil Gandhi, Arnold P Boedihardjo, Crystal Chen, and Susan Frankenstein. 2016. RPM: Representative Pattern Mining for Efficient Time Series Classification. In *Proceedings of the 19th International Conference on Extending Database Technology*.
- [40] Michael Wojnowicz, Glenn Chisholm, Brian Wallace, Matt Wolff, Xuan Zhao, and Jay Luan. 2017. SUSPEND: Determining software suspiciousness by non-stationary time series modeling of entropy signals. *Expert Systems with Applications* 71 (2017), 301–318.
- [41] David H Wolpert. 1992. Stacked generalization. *Neural networks* 5, 2 (1992), 241–259.
- [42] Xiaoke Xu, Jie Zhang, and Michael Small. 2008. Superfamily phenomena and motifs of networks induced from time series. *Proceedings of the National Academy of Sciences* 105, 50 (2008), 19601–19605.
- [43] Jianbo Yang, Minh Nhut Nguyen, Phyo Phyo San, Xiaoli Li, and Shonali Krishnaswamy. 2015. Deep Convolutional Neural Networks on Multichannel Time Series for Human Activity Recognition.. In *IJCAI*. 3995–4001.
- [44] Lexiang Ye and Eamonn Keogh. 2009. Time series shapelets: a new primitive for data mining. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 947–956.

Counting Edges with Target Labels in Online Social Networks via Random Walk

Yang Wu

Chinese University of Hong Kong
Shatin, Hong Kong, China
yangwu@cse.cuhk.edu.hk

Ada Wai-Chee Fu

Chinese University of Hong Kong
Shatin, Hong Kong, China
adafu@cse.cuhk.edu.hk

Cheng Long

Queen's University Belfast
Belfast, United Kingdom
cheng.long@qub.ac.uk

Zitong Chen

Chinese University of Hong Kong
Shatin, Hong Kong, China
ztchen@cse.cuhk.edu.hk

ABSTRACT

Online social networks (OSNs) embed a rich set of information that could be used in a few fields. Extensive research has been done on estimating graph properties such as counts of wedges and triangles in OSNs. While these graph properties which are defined based on the structural information only are useful at a coarse level, they are not sufficient in applications where finer-grained information is desired. In this paper, we study a problem of estimating a type of graph property, namely the count of edges, refined by the labels of the users, which are usually available in users' profiles, and serves as finer-grained information. Existing solutions for estimating graph properties pay no attention on users' labels and thus they are not suitable for many real world applications. We develop two algorithms for the problem, each of which samples a set of edges or nodes via a random walk process and construct estimators based on the sampled edges or nodes. Theoretical analysis on the accuracy guarantees of our algorithms and extensive experiments based on real datasets verify that our algorithms are superior over baseline algorithms.

1 INTRODUCTION

Online social networks (OSN) is very commonly used in real life and it embeds a rich set of information that would be useful in applications from different fields such as social community, marketing business, political campaigns, etc. People are interested in knowing some information of graph properties such as counts of wedges, triangles, cliques, and k -node structures etc. embedded in OSNs. In the literature, researchers have studied problems of estimating degree distribution [7, 14, 16], clustering coefficient [11], graph size [11, 13] and graphlet statistics [5, 21]. These graph properties are usually defined based on the structural information, e.g., a triangle is a triplet of three nodes which are connected with one another via links.

We notice that graph properties based on the structure information correspond to information at a coarse level only, which may not be sufficient in some applications. For example, if an education institution considers to introduce a new Spanish course in Hong Kong, the most important step is to determine whether there are enough potential users who are likely to take this course in Hong Kong. One simple but efficient way is to estimate the number of links/friendships between a user living in Hong Kong and another user living in Spain in OSNs. The reason is that if a

user has Spanish friends, then it is likely that he/she will be interested in learning Spanish. Another example is that estimating the number of links/friendships between a user living in China and another user living in Austria in an OSN is an indicator of how many people from Austria and those from China interact with each other. Such information is very useful for airlines in web marketing and advertising, e.g., it could be used for decision making about whether or not to launch a new flight route between China and Austria. Thus, graph properties refined by some feature/label information of users (which are available in user's profiles in many cases) correspond to finer-grained information and could be highly valuable in real world application.

Motivated by this, we propose to estimate graph properties refined by users' labels. In this paper, we focus on one type of graph properties, namely the number of edges with some target labels. Specifically, given two target labels, we say that an edge is a *target edge* if one node of the edge has one target label and the other node has the other target label. For example, in the example of estimating the number of links between users living in Spain and those living in Hong Kong, "Spain" and "Hong Kong" could be used as two target labels and an edge between a user living in Spain and another user living in Hong Kong corresponds to a target edge. Then, the problem studied in this paper is to estimate the number of target edges for two given target labels.

Existing solutions of estimating graph properties (based on the structural information only) do not pay any attention to users' labels and thus, they could not be used for our problem of estimating graph properties (based on both the structural information and the information of users' labels). Another branch of studies that is related to ours is labeled graph mining, such as graph classification [20], subgraph mining [3, 4, 9] and label prediction [24]. However, these solutions all assume full access to the graph, which is not true when dealing with OSNs as we do in this paper since OSNs are only accessible via provided APIs [11, 13].

To solve the problem of estimating the number of target edges, we develop two algorithms, namely, NeighborSample and NeighborExploration, both of which are based on a random walk on the graph and sample a set of edges or nodes. NeighborSample samples a set of k edges with k iterations, at each iteration it samples one edge by sampling a user via a random walk process and then sampling a neighbor of this user. NeighborExploration samples a set of nodes with k iterations via a random walk process and also explores all neighbors of each sampled node and records the number of target edges incident to the sampled node, if the sampled node involves a target label (with the purpose of sampling target edges with higher probabilities). Then, based on the

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

sample set, we construct unbiased estimators using some statistical techniques. For each estimator, we conduct some theoretical analysis on the relationship between the number of samples and the corresponding accuracy guarantees.

In summary, our main contributions are as follows. First, we propose to estimate graph properties refined by users' labels in OSNs, which, to the best of our knowledge, is the first attempt to do so. Second, we develop two algorithms for estimating the number of edges with target labels and provide theoretical analysis on the accuracy guarantees. Third, we conducted extensive experiments which verified that the algorithms developed in this paper are superior over baseline algorithms.

This paper is organized as follows. Section 2 reviews the related work. Section 3 provides some preliminaries and the problem definition. Section 4 introduces our proposed algorithms for estimating the number of edges with target labels. Section 5 presents the experiments and Section 6 concludes the paper and gives a few directions for future study.

2 RELATED WORK

In the literature, OSNs with restricted access and labeled graphs with full access are both popular topics, but to our knowledge, there has been no study about labeled graphs with restricted access, which we study in this paper.

A lot of work has been done on online social networks with restricted access and most existing work is based on random walk methods, which have been used for estimating degree distribution [7, 14, 16], clustering coefficients [11] and graph size [11, 13]. In [14], the authors introduced a non-backtracking random walk method, which is more efficient than traditional random walk for estimating degree distribution. In [11], the authors proposed simple but efficient sampling algorithms for estimating the clustering coefficient and the graph size via simple random walk. In addition, node pairs sampling in OSNs has also been studied in [22]. Recently, Chen et al. [5] proposed the state-of-the-art random walk based algorithms for graphlets statistics using the concepts of subgraph relationship graphs and expanded Markov Chain.

Labels in graph are widely used in many applications, which has attracted much attention from researchers and a considerable amount of work has been done on labeled graphs. Particularly, it has been studied in the area of subgraph mining a lot. In [9], the authors studied the problem of mining frequent neighbourhood pattern in labeled graphs and in [4], a method for mining significant connected subgraph in labeled graphs is proposed. Anchuri et al. consider the difference between labels as a cost and introduce an algorithm for mining approximate subgraph patterns with label cost [3]. Labels are also used in graph classification. In [20], the authors proposed an algorithm for classifying labeled nodes based on structural neighbourhood. Recently, Ye et al. [24] considered a new scenario where only very few vertices have labels compared to large amounts of unlabeled vertices, and proposed an algorithm which leverages the limited user information and friendship network wisely to infer the labels of unlabeled users in OSNs.

3 PRELIMINARIES AND PROBLEM DEFINITION

In this section, we first introduce some basic notations, and then give a formal definition of the problem.

An OSN is represented as an undirected graph $G(V, E)$, where a user corresponds to a node in V and a friendship between two users corresponds to an edge in E . For each user u in G , we denote u 's degree by $d(u)$, i.e., $d(u)$ corresponds to the number of u 's friends. Each user/node in V has a set of labels such as this user's gender, profession, living country etc. which could be found out in most cases by checking users' profile. For an edge (u, v) , we define its label as a pair of two labels, one is a label of u and the other is a label of v .

Let t_1 and t_2 be two target labels. We say that an edge (u, v) is a *target edge* if the edge has the pair of t_1 and t_2 as one of its labels, i.e., either u has t_1 and v has t_2 , or v has t_1 and u has t_2 . We say that (t_1, t_2) is the target edge label. Let F be the number of target edges. In this paper, we study the problem of estimating F with the following assumptions: (1) we have no full access to the graph $G(V, E)$ but only some limited access via APIs each of which can be used to retrieve the list of friends/neighbors of a given user; (2) the information of $|V|$ and $|E|$ is available as prior knowledge (this is reasonable since this information can be often obtained from the OSN owner's reports or Internet, and in case that such information is not publicly available, some existing methods such as [11] and [23] could be used to estimate $|V|$ and $|E|$, respectively).

We also derive the bound on the sample size which can achieve an (ϵ, δ) -approximation estimation of F . The \hat{F} which satisfies the following equation is an (ϵ, δ) -approximation estimation of F .

$$P[(1 - \epsilon)F < \hat{F} < (1 + \epsilon)F] \geq 1 - \delta \quad (1)$$

4 ESTIMATORS OF NUMBER OF EDGES WITH TARGET LABEL

We propose to estimate the number of edges with target labels by first sampling a set of edges or nodes and then constructing an (un-biased) estimator based on the set of sampled edges or nodes. In the following, we introduce two algorithms, one for sampling edges and the other for sampling nodes, both of which are based on a random walk on the graph. The first one, as presented in Section 4.1, is called *NeighborSample* and samples a set of k edges with k iterations at each of which it samples one edge by sampling a user via a random walk process and then sampling a neighbor of this user. The second one, as presented in Section 4.2, is called *NeighborExploration* and samples a set of nodes with k iterations via a random walk process and also explores all neighbors of each sampled node and records the number of target edges incident to the sampled node, if the sampled node involves a target label (with the purpose of sampling target edges with higher probabilities).

For each sampling algorithm, we use Hansen-Hurwitz estimator [10], Horvitz-Thompson estimator [12] and Re-weighted estimator [17] to estimate the number of target edges. Hansen-Hurwitz estimator and Horvitz-Thompson estimator are two simple and widely used estimator when samples are sampled with unequal probabilities, and Re-weighted estimator is an estimator based on the Hansen-Hurwitz estimator.

4.1 Estimation Based on NeighborSample

4.1.1 Sampling Process. NeighborSample samples a set S of k edges with k iterations. At each iteration, it samples an edge by sampling a user u via simple random walk first and then randomly picking one of u 's neighbors, says v (i.e., (u, v) corresponds to the edge sampled at this iteration). The pseudo-code of NeighborSample is presented in Algorithm 1.

Algorithm 1 NeighborSample

Require: an online social network $G(V, E)$ accessible via APIs

- 1: $S \leftarrow \emptyset$
 - 2: **for** $i: 1 \leftarrow k$ **do**
 - 3: Sample a user u_i via simple random walk
 - 4: Sample a neighbor v_i of u_i randomly
 - 5: $S \leftarrow S \cup (u_i, v_i)$
 - 6: **end for**
 - 7: **Return** S
-

Based on the sampling process, we construct two different estimators, one is based on the *Hansen-Hurwitz Estimator* and the other is based on the *Horvitz-Thompson Estimator*.

4.1.2 Hansen-Hurwitz Estimator. We define X_i ($1 \leq i \leq k$) to be the edge sampled at the i^{th} iteration of the sampling process. Consider the distribution of X_i . First, X_i could be any edge $(u, v) \in E$. Second, an edge (u, v) is sampled if and only if the following two events happen: (E1) u is sampled by the random walk (line 3 in Algorithm 1) and then v , as one of u 's neighbors, is picked (line 4 in Algorithm 1) and (E2) v is sampled by the random walk (line 3 in Algorithm 1) and then u , as one of v 's neighbors, is picked (line 4 in Algorithm 1). Third, the probability of event E1 is equal to $\frac{d(u)}{2|E|} \cdot \frac{1}{d(u)} = \frac{1}{2|E|}$ (the probability that u is sampled by the random walk is equal to $\frac{d(u)}{2|E|}$ according to the stationary distribution of a random walk [8, 18] and the probability that one specific neighbor of u is sampled is equal to $\frac{1}{d(u)}$) and so is that of event E2. Therefore, the probability that X_i corresponds to any edge (u, v) , denoted by $\pi(X_i = (u, v))$, is equal to $\frac{1}{2|E|} + \frac{1}{2|E|} = \frac{2}{2|E|} = \frac{1}{|E|}$, i.e., X_i corresponds to a uniform sample from the set of edges.

Now, consider $\frac{I(X_i)}{\pi(X_i)}$ which is also a random variable, where $I(X_i)$ is an indicator function and $I(X_i) = 1$ if X_i is one target edge, and 0 otherwise. We deduce that $E[\frac{I(X_i)}{\pi(X_i)}] = \sum_{X_i \in E} I(X_i) = F$. Based upon on this, we construct an estimator of F as $\frac{I((u_i, v_i))}{\pi(X_i = (u_i, v_i))} = |E| \cdot I((u_i, v_i))$, which could be verified to be unbiased. Since in each iteration, we can construct such an estimator, we use the average of these estimators as the final estimator, which is presented as follows.

$$\hat{F} = \frac{1}{k} \sum_{1 \leq i \leq k} |E| \cdot I(X_i) = \frac{1}{k} \sum_{1 \leq i \leq k} |E| \cdot I((u_i, v_i)) \quad (2)$$

This estimator corresponds to a Hansen-Hurwitz estimator [10].

Implementation. A straightforward implementation of the NeighborSample sampling process as shown in Algorithm 1 would perform k random walk processes. We note that performing a random walk process is costly since it needs to walk for some enough steps (which corresponds to the *mixing time* [6] of the random walk) in order to achieve the stationary distribution, where each walk from one user to one of her neighbors requires to issue an API call. Fortunately, we observe that the above estimator does not require to sample edges (u_i, v_i) ($1 \leq i \leq k$) independently, and thus we propose to sample all these edges using a *single* random walk process. Specifically, it performs a enough number of simple random walk steps first (i.e., the mixing time is achieved) and then continues to walk for k steps further, each via an edge. At the end, it picks those edges it walks through at the last k steps as the sampled edges. In this way, k edges are sampled via only a single random process and as could be verified, the probability that one sampled edge corresponds

to a specific edge in the graph is still equal to $\frac{1}{|E|}$ and thus the estimator constructed above is still valid.

Analysis. In this part, we derive theoretical results on the number of edges to sample in order to achieve some pre-set accuracy guarantee.

THEOREM 4.1. *Let $1 > \delta > 0$, $\epsilon \leq 1$ and $k \geq 1$. Sampling k edges in NeighborSample will return an (ϵ, δ) -approximation estimation of F by the Hansen-Hurwitz estimator, if*

$$k \geq \frac{\sum_{X \in E} |E| \cdot I(X) - F^2}{\epsilon^2 \cdot F^2 \cdot \delta}$$

PROOF. Since $F = E[\frac{I(X_i)}{\pi(X_i)}]$, if $\frac{1}{k} \sum_{i=1}^k \frac{I(X_i)}{\pi(X_i)}$ is an (ϵ, δ) -approximation estimation of $E[\frac{I(X_i)}{\pi(X_i)}]$, then \hat{F} is also an (ϵ, δ) -approximation estimation of F .

Let $Y = \frac{1}{k} \sum_{i=1}^k \frac{I(X_i)}{\pi(X_i)}$, then $E[Y] = F$ and $Var[Y] = \frac{1}{k} (E[\frac{I(X_i)^2}{\pi(X_i)^2}] - E[\frac{I(X_i)}{\pi(X_i)}]^2) = \frac{1}{k} \cdot \sum_{X \in E} \frac{I(X)}{\pi(X)} - F^2$ and $E[Y] = F$. By Chebyshev's inequality (see Appendix A), we obtain the bound of k for achieving (ϵ, δ) -approximation.

$$k \geq \frac{\sum_{X \in E} |E| \cdot I(X) - F^2}{\epsilon^2 \cdot F^2 \cdot \delta} \quad \square$$

4.1.3 Horvitz-Thompson Estimator. For each edge $e = (u, v) \in E$, we define a new indicator function $H(e \in S)$ such that $H(e \in S)$ is equal to 1 if e is sampled by the NeighborSample sampling process once or multiple times, and 0 otherwise. We define $Pr(e)$ as the probability that an edge e is sampled in at least one iteration of the NeighborSample sampling process, i.e., $e \in S$. Consider $Pr(e)$ for a specific edge e , we observe that the event that e is not sampled happens if and only if all following k events happen: e is not sampled in the i^{th} iteration for $1 \leq i \leq k$. Considering that the probability that each of these events happens is equal to $(1 - \frac{1}{|E|})$ and these events are independent, we know that the probability that e is not sampled in any of iterations is equal to $(1 - \frac{1}{|E|})^k$, which further implies that the probability that e is sampled in at least one of the iterations, i.e., $Pr(e)$, is equal to $1 - (1 - \frac{1}{|E|})^k$.

Then, we construct an estimator of F as follows.

$$\hat{F} = \sum_{e \in E} \frac{I(e)}{Pr(e)} H(e \in S) = \sum_{e \in E} \frac{I(e)}{1 - (1 - \frac{1}{|E|})^k} H(e \in S) \quad (3)$$

Next we show how this estimator is derived and that it is unbiased. Define $\{S_1, S_2, \dots, S_m\}$ as the collection of all possible sample sets each of which contains k edges from the edge set E . Let $P(S_i)$ be the probability that we get set S_i as the sample set in NeighborSample process. Let S be a random set from $\{S_1, S_2, \dots, S_m\}$, we have

$$\begin{aligned} E[\sum_{e \in E} \frac{I(e)}{Pr(e)} H(e \in S)] &= \sum_{j=1}^m P(S_j) \sum_{e \in E} \frac{I(e)}{Pr(e)} H(e \in S_j) \\ &= \sum_{e \in E} \frac{I(e)}{Pr(e)} \sum_{j=1}^m P(S_j) H(e \in S_j) \\ &= \sum_{e \in E} \frac{I(e)}{Pr(e)} Pr(e) = \sum_{e \in E} I(e) = F \end{aligned} \quad (4)$$

So the estimator $\hat{F} = \sum_{e \in E} \frac{I(e)}{Pr(e)} H(e \in S)$ is an unbiased estimator.

Implementation. Different from the case of the Hansen-Hurwitz estimator in Section 4.1.2, the Horvits-Thompson estimator requires that the edges sampled in different iterations are independent. With the implementation in Section 4.1.2, the edges sampled are not independent since the edge sampled at the current iteration is adjacent to the one sampled at the last iteration. To meet the independence requirement, we adopt an existing strategy [11], which is to use those vertices (and edges) which are

sampled far away from each other by a certain number r of steps in the random walk process, (in this way, every two sampled edges could be regarded as approximately independent sampled edges, and following [11], we set r as $2.5\%k$) in our experiments.

Analysis. In this part, we derive some theoretical results on the number of edges to sample in order to achieve some pre-set accuracy guarantee.

THEOREM 4.2. *Let $1 > \delta > 0$, $\epsilon \leq 1$ and $k \geq 1$. Sampling k edges in NeighborSample will return an (ϵ, δ) -approximation estimation of F by the Horvitz-Thompson estimator, if*

$$k \geq \max_{e \in E} \log \frac{I(e)^2 + B}{B} / \log \frac{1}{A(e)}$$

where $A(e) = 1 - \frac{1}{|E|}$ and $B = \delta \epsilon^2 \cdot F^2 / |E|$.

PROOF. Let $\pi_e = \frac{1}{|E|}$ be the probability of sampling edge e in one NeighborSample process. In [12], it has been proved that the variance of the Horvitz-Thompson estimator is

$$\begin{aligned} \text{Var}[\widehat{F}] &= E[\widehat{F}^2] - E[\widehat{F}]^2 \\ &= \sum_{e_1 \in E} \sum_{e_2 \in E} \frac{I(e_1)I(e_2)}{\Pr(e_1)\Pr(e_2)} E[H(e_1 \in S)H(e_2 \in S)] \\ &\quad - \sum_{e_1 \in E} \sum_{e_2 \in E} \frac{I(e_1)I(e_2)}{\Pr(e_1)\Pr(e_2)} E[H(e_1 \in S)]E[H(e_2 \in S)] \\ &= \sum_{e_1 \in E} \sum_{e_2 \in E} \frac{I(e_1)I(e_2)}{\Pr(e_1)\Pr(e_2)} \text{Cov}(H(e_1 \in S), H(e_2 \in S)) \end{aligned} \quad (5)$$

Since $\text{Cov}(H(e_1 \in S), H(e_2 \in S)) = \Pr(e_1, e_2) - \Pr(e_1)\Pr(e_2)$, where $\Pr(e_1, e_2) = \Pr(e_1) + \Pr(e_2) - (1 - (1 - \pi_{e_1} - \pi_{e_2})^k)$ for $e_1 \neq e_2$ and $\text{Cov}(H(e_1 \in S), H(e_2 \in S)) = \Pr(e_1)(1 - \Pr(e_1))$ for $e_1 = e_2$, so we have

$$\begin{aligned} \text{Var}[\widehat{F}] &= \sum_{e_1 \in E} \left(\frac{1 - \Pr(e_1)}{\Pr(e_1)} \right) I(e_1)^2 + \\ &\quad \sum_{e_1 \in E} \sum_{e_2 \in E, e_2 \neq e_1} \left(\frac{\Pr(e_1) + \Pr(e_2)}{\Pr(e_1)\Pr(e_2)} + \right. \\ &\quad \left. - \frac{\{1 - (1 - \pi_{e_1} - \pi_{e_2})^k\} - \Pr(e_1)\Pr(e_2)}{\Pr(e_1)\Pr(e_2)} \right) I(e_1)I(e_2) \end{aligned} \quad (6)$$

The second term in the right hand side of Equation (6) can be simplified as

$$\begin{aligned} &\sum_{e_1 \in E} \sum_{e_2 \in E, e_2 \neq e_1} \left(\frac{(1 - \pi_{e_1} - \pi_{e_2})^k}{\Pr(e_1)\Pr(e_2)} \right. \\ &\quad \left. - \frac{(1 - \pi_{e_1} - \pi_{e_2} + \pi_{e_1}\pi_{e_2})^k}{\Pr(e_1)\Pr(e_2)} \right) I(e_1)I(e_2) \end{aligned} \quad (7)$$

since $\pi_{e_1} = \pi_{e_2} = 1/|E| \geq 0$, it is obvious that $1 - \pi_{e_1} - \pi_{e_2} \leq 1 - \pi_{e_1} - \pi_{e_2} + \pi_{e_1}\pi_{e_2}$. As a result, the term 7 is also negative, so we can ignore this term. By Chebyshev's inequality (see Appendix A), we have

$$\sum_{e \in E} \left(\frac{(1 - \pi_e)^k}{1 - (1 - \pi_e)^k} \right) I(e)^2 \leq \delta \epsilon^2 \cdot F^2 \quad (8)$$

so if

$$\left(\frac{(1 - \pi_e)^k}{1 - (1 - \pi_e)^k} \right) I(e)^2 \leq \delta \epsilon^2 \cdot F^2 / |E| \quad (9)$$

holds for each $e \in E$, then Equation (8) also holds.

Define $A(e) = 1 - \pi_e$ and $B = \delta \epsilon^2 \cdot F^2 / |E|$, then we obtain the bound of k for achieving (ϵ, δ) -approximation.

$$k \geq \max_{e \in E} \log \frac{I(e)^2 + B}{B} / \log \frac{1}{A(e)} \quad (10)$$

□

4.2 Estimation Based on NeighborExploration

4.2.1 Sampling Process. NeighborExploration samples a set S of nodes with k iterations for a given integer k . At each iteration, it first samples a node by a random walk process and then explores *all* edges incident to u if u involves a target label. The

Algorithm 2 NeighborExploration

Require: an online social network $G(V, E)$ accessible via APIs
Require: a pair of two target labels t_1 and t_2

- 1: $S \leftarrow \emptyset$
 - 2: **for** $i: 1 \leftarrow k$ **do**
 - 3: Sample a user u_i via simple random walk
 - 4: **if** u_i has label t_1 or label t_2 **then**
 - 5: Explore all the neighbors of u_i and compute the number of target edges incident to u_i and we use function T to record the mapping from u_i to the number of target edges incident to u_i .
 - 6: **end if**
 - 7: $S \leftarrow S \cup u_i$
 - 8: **end for**
 - 9: **Return** S and function T
-

rationale of exploring all neighbors of the user u is that once we know that u has a target label, the probability that we can find a target edge incident to u would be relatively high since one of the two target labels has been covered already. The pseudo-code of NeighborExploration is presented in Algorithm 2.

4.2.2 Hansen-Hurwitz Estimator. We define a random variable Y_i to be the node sampled at i^{th} iteration of NeighborExploration sampling process. Y_i could be any user $u \in V$ and the probability that Y_i corresponds to a specific user u , denoted by $\pi(Y_i = u)$, is equal to $\frac{d(u)}{2|E|}$ which is based on the stationary distribution of a simple random walk. We define $T(Y_i)$ as the number of target edges incident to Y_i , which also corresponds to a random variable and could be computed when all neighbors of Y_i are explored after we sample Y_i in the random walk process.

Now, consider $\frac{T(Y_i)}{\pi(Y_i)}$ which is also a random variable. We deduce that $E[\frac{T(Y_i)}{\pi(Y_i)}] = \sum_{u \in V} T(u) = 2 \cdot F$. Based upon on this, we construct an estimator of F as $\frac{T(Y_i)}{2 \cdot \pi(Y_i)}$ which could be easily verified to be unbiased. Since in each iteration, we can construct such an estimator, we use the average of these estimators as the final estimator, which is presented as follows.

$$\hat{F} = \frac{1}{k} \sum_{1 \leq i \leq k} \frac{T(Y_i)}{2 \cdot \pi(Y_i)} = \frac{1}{k} \sum_{1 \leq i \leq k} \frac{|E| \cdot T(u_i)}{d(u_i)} \quad (11)$$

This estimator corresponds to a Hansen-Hurwitz estimator [10].

Implementation. Same as the case in Section 4.1.2, the estimator here does not require that the sampled nodes are independent. As a result, it can sample all nodes via a single random process by performing an enough number of simple random walk steps first (i.e., the mixing time is achieved) and then continuing to walk for k steps further. At each of the last k steps, it checks whether the current user u involves a target label. If so, it explores all edges incident to this user, and records $T(u)$.

Analysis. In this part, we derive some theoretical results on the number of nodes to sample in order to achieve some pre-set accuracy guarantee.

THEOREM 4.3. *Let $1 > \delta > 0$, $\epsilon \leq 1$ and $k \geq 1$. Sampling k nodes in NeighborExploration will return an (ϵ, δ) -approximation estimation of F by the Hansen-Hurwitz estimator, if*

$$k \geq \frac{\sum_{u \in V} \frac{2|E| \cdot T(u)^2}{d(u)} - 4F^2}{4\epsilon^2 \cdot F^2 \cdot \delta}$$

PROOF. Since $F = \frac{1}{2}E[\frac{T(Y_i)}{\pi(Y_i)}]$, if $\frac{1}{k}\sum_{i=1}^k \frac{T(Y_i)}{\pi(Y_i)}$ is an (ϵ, δ) -approximation estimation of $E[\frac{T(Y_i)}{\pi(Y_i)}]$, then \hat{F} is also an (ϵ, δ) -approximation estimation of F .

Let $X = \frac{1}{k}\sum_{i=1}^k \frac{T(Y_i)}{\pi(Y_i)}$, then $E[X] = 2F$ and $Var[X] = \frac{1}{k}(E[\frac{T(Y)^2}{\pi_Y^2}] - E[\frac{T(Y)}{\pi_Y}]^2) = \frac{1}{k} \cdot \sum_{u \in N} \frac{T(u)^2}{\pi_u} - 4F^2$. By Chebyshev's inequality (see Appendix A), we obtain the bound of k for achieving (ϵ, δ) -approximation.

$$k \geq \frac{\sum_{u \in V} \frac{2|E| \cdot T(u)^2}{d_u} - 4F^2}{4\epsilon^2 \cdot F^2 \cdot \delta} \quad (12)$$

□

4.2.3 Horvitz-Thompson Estimator. We define an indicator function $H(u \in S)$ such that $H(u \in S)$ is equal to 1 if u is sampled by the NeighborExploration sampling process once or multiple times, and 0 otherwise. S is the sample set obtained from the NeighborExploration sampling process. Then, we define $Pr(u)$ as the probability that a node u is sampled in at least one iteration of the NeighborExploration sampling process, i.e., $u \in S$. Consider $Pr(u)$ for a specific node u . We observe that the event that u is not sampled happens if and only if all following k events happen: u is not sampled in the i^{th} iteration for $1 \leq i \leq k$. Considering that the probability that one of these events happens is equal to $(1 - \frac{d(u)}{2|E|})$, and these events are independent, we know that the probability that u is not sampled in any of iterations is equal to $(1 - \frac{d(u)}{2|E|})^k$, which further implies that the probability that u is sampled in at least one of the iteration, i.e., $Pr(u)$ is equal to $(1 - (1 - \frac{d(u)}{2|E|})^k)$.

Then, we construct a Horvitz-Thompson estimator of the number of target edges as follows.

$$\hat{F} = \frac{1}{2} \sum_{u \in V} \frac{T(u)}{Pr(u)} H(u \in S) = \frac{1}{2} \sum_{u \in V} \frac{T(u)}{1 - (1 - \frac{d(u)}{2|E|})^k} H(u \in S) \quad (13)$$

This estimator could be verified to be unbiased similarly as it is done for the Horvitz-Thompson estimator based on Neighbor-Sample in Section 4.1.3.

Implementation. Also, the Horvitz-Thompson estimator requires that the nodes sampled in different iterations are independent. With the implementation in Section 4.2.2, the nodes sampled are not independent since the node sampled at the current iteration is adjacent to the one sampled at the last iteration. To meet the independence requirement, we use the same strategy introduced in 4.1.3, which is to use those nodes which are sampled far away from each other by a certain number r of steps in the random walk process, and we set r as $2.5\%k$.

Analysis. In this part, we derive theoretical results on the number of nodes to sample in order to achieve some pre-set accuracy guarantee.

THEOREM 4.4. *Let $1 > \delta > 0$, $\epsilon \leq 1$, and $k \geq 1$. Sampling k nodes in NeighborExploration will return an (ϵ, δ) -approximation estimation by the Horvitz-Thompson estimator, if*

$$k \geq \max_{y \in V} \log \frac{T(y)^2 + B}{B} / \log \frac{1}{A(y)}$$

where $A(y) = 1 - \pi_y$ and $B = 4\delta\epsilon^2 \cdot F^2 / |V|$.

PROOF. Let $\pi_y = \frac{d_y}{2|E|}$ be the probability of sampling node y in the random walk process. In [12], it has been proved that the

variance of the Horvitz-Thompson estimator is

$$Var[\hat{F}] = \frac{1}{4} \{ \sum_{y \in V} (\frac{1 - Pr(y)}{Pr(y)}) T(y)^2 + \sum_{y \in V} \sum_{z \in V, z \neq y} (\frac{Pr(y) + Pr(z)}{Pr(y)Pr(z)} - \frac{-\{1 - (1 - \pi_y - \pi_z)^k\} - Pr(y)Pr(z)}{Pr(y)Pr(z)}) T(y)T(z) \} \quad (14)$$

The second term in the right hand side of Equation (14) can be simplified as

$$\sum_{y \in V} \sum_{z \in V, z \neq y} (\frac{(1 - \pi_y - \pi_z)^k}{Pr(y)Pr(z)} - \frac{(1 - \pi_y - \pi_z + \pi_y \pi_z)^k}{Pr(y)Pr(z)}) T(y)T(z) \quad (15)$$

since $\pi_y = d_y / (2|E|) \geq 0$, it is obvious that $1 - \pi_y - \pi_z \leq 1 - \pi_y - \pi_z + \pi_y \pi_z$. As a result, term (15) is negative, so we can ignore this term. Then By Chebyshev's inequality (see Appendix A), we have

$$\sum_{y \in V} (\frac{(1 - \pi_y)^k}{1 - (1 - \pi_y)^k}) T(y)^2 \leq 4\delta\epsilon^2 \cdot F^2 \quad (16)$$

so if

$$(\frac{(1 - \pi_y)^k}{1 - (1 - \pi_y)^k}) T(y)^2 \leq 4\delta\epsilon^2 \cdot F^2 / |V| \quad (17)$$

holds for each $y \in V$, then Equation (16) also holds.

Define $A(y) = 1 - \pi_y$ and $B = 4\delta\epsilon^2 \cdot F^2 / |V|$, we can get the bound of k ,

$$k \geq \max_{y \in V} \log \frac{T(y)^2 + B}{B} / \log \frac{1}{A(y)} \quad (18)$$

□

4.2.4 Re-Weighted Estimator. Based on the NeighborExploration sampling process, the nodes are sampled with non-uniform probabilities, which is different from the case based on the NeighborExploration sampling process. This makes it possible to construct a *Re-weighted* estimator [17] as follows.

$$\hat{F} = \frac{\sum_{i=1}^k T(u_i) / d(u_i)}{2 \sum_{i=1}^k 1 / d(u_i)} \cdot |V| \quad (19)$$

Here, u_i corresponds to the user sampled via the random walk process in i^{th} iteration. It was known that the Re-weighted estimator can be interpreted using the *importance sampling* (IS) framework [17]. Specifically, instead of sampling nodes from the target distribution (i.e., the uniform distribution), the IS framework samples edges from a different and easily implemented trial distribution (i.e., the stationary distribution of a random walk process). According to the IS framework, the importance weight of a user u is given by $\frac{1/|V|}{d(u)/2|E|} \propto 1/d(u)$, which meets the definition in Equation (19), where $1/|V|$ corresponds to the probability based on the target distribution and $d(u)/2|E|$ corresponds the probability based on trial distribution.

Implementation. Same as the Hansen-Hurwitz estimator in Section 4.2.2, the Re-weighted estimator constructed here does not require that the nodes sampled are independent, and thus the implementation described in Section 4.2.2, which samples all nodes with one single random walk process, could be applied.

Analysis. In this part, we derive theoretical results on the number of nodes to sample in order to achieve some pre-set accuracy guarantee.

THEOREM 4.5. *Let $1 > \delta > 0$, $\epsilon \leq 1$ and $k \geq 1$. Sampling k nodes in NeighborExploration will return an (ϵ, δ) -approximation estimation of F the by the Re-Weighted estimator, if*

$$k \geq \max \{ 18 \frac{\sum_{y \in V} \frac{T(y)^2}{\pi_y} - 4F^2}{\epsilon^2 \cdot 4F^2 \cdot \delta}, 18 \frac{\sum_{y \in V} \frac{1}{\pi_y} - |V|^2}{\epsilon^2 \cdot |V|^2 \cdot \delta} \}$$

PROOF. Let Y be a random node sampled from one random walk step and $\pi_Y = \frac{d(Y)}{2|E|}$ be the probability of sampling node Y in the random walk process. Since $F = E[\frac{T(Y)}{\pi_Y}]/E[\frac{1}{\pi_Y}] \cdot \frac{|V|}{2}$ and $\widehat{F} = \frac{E[\frac{T(Y)}{\pi_Y}]/E[\frac{1}{\pi_Y}]}{E[\frac{T(Y)}{\pi_Y}]/E[\frac{1}{\pi_Y}]} \cdot \frac{|V|}{2}$, if $E[\frac{T(Y)}{\pi_Y}]/E[\frac{1}{\pi_Y}]$ is an (ϵ, δ) -approximation estimation of $E[\frac{T(Y)}{\pi_Y}]/E[\frac{1}{\pi_Y}]$, then \widehat{F} is also an (ϵ, δ) -approximation estimation of F .

Assume that $E[\frac{T(Y)}{\pi_Y}]$ and $E[\frac{1}{\pi_Y}]$ are $(\epsilon/3, \delta/2)$ -approximation estimations of $E[\frac{T(Y)}{\pi_Y}]$ and $E[\frac{1}{\pi_Y}]$ respectively. Let $A = \frac{E[\frac{T(Y)}{\pi_Y}]/E[\frac{1}{\pi_Y}]}{E[\frac{T(Y)}{\pi_Y}]/E[\frac{1}{\pi_Y}]}$ and $B = \frac{E[\frac{T(Y)}{\pi_Y}]/E[\frac{1}{\pi_Y}]}{E[\frac{1}{\pi_Y}]}$, then we have $P[A/B > \frac{1-\epsilon/3}{1+\epsilon/3}] > 1 - \epsilon, A/B < \frac{1+\epsilon/3}{1-\epsilon/3} > 1 + \epsilon \geq (1 - \delta/2)^2 > 1 - \delta$. So $E[\frac{T(Y)}{\pi_Y}]/E[\frac{1}{\pi_Y}]$ is an (ϵ, δ) -approximation estimation of $E[\frac{T(Y)}{\pi_Y}]/E[\frac{1}{\pi_Y}]$.

Let $X = \frac{1}{k} \sum_{i=1}^k \frac{T(y_i)}{\pi_{y_i}}$ and $Z = \frac{1}{k} \sum_{i=1}^k \frac{1}{\pi_{y_i}}$. By Chebyshev's inequality in Appendix A, we have

$$Pr[|X - E[X]| > \epsilon/3 \cdot E[X]] \leq \frac{Var[X]}{(\epsilon/3 \cdot E[X])^2} \leq \delta/2 \quad (20)$$

$$Pr[|Z - E[Z]| > \epsilon/3 \cdot E[Z]] \leq \frac{Var[Z]}{(\epsilon/3 \cdot E[Z])^2} \leq \delta/2 \quad (21)$$

Since $E[X] = 2F, Var[X] = \frac{1}{k} (E[\frac{T(Y)^2}{\pi_Y^2}] - E^2[\frac{T(Y)}{\pi_Y}]) = \frac{1}{k} (\sum_{y \in V} \frac{T(y)^2}{\pi_y} - 4F^2), E[Z] = |V|$ and $Var[Z] = \frac{1}{k} (E[\frac{1}{\pi_Y^2}] - E^2[\frac{1}{\pi_Y}]) = \frac{1}{k} (\sum_{y \in V} \frac{1}{\pi_y} - |V|^2)$, we have

$$k \geq \max\left\{18 \frac{\sum_{y \in V} \frac{T(y)^2}{\pi_y} - 4F^2}{\epsilon^2 \cdot 4F^2 \cdot \delta}, 18 \frac{\sum_{y \in V} \frac{1}{\pi_y} - |V|^2}{\epsilon^2 \cdot |V|^2 \cdot \delta}\right\} \quad (22)$$

□

5 EXPERIMENTAL RESULTS

5.1 Experimental Set-up

Datasets. We used 5 real datasets which are publicly available and widely used in previous work [5, 11, 13] as shown in Table 1. In the experiment, we simulate the scenario where we only have accesses to the graphs via APIs. In each network, we remove the directions of edges, self-loops and multi-edges. We use the largest connected component for each network (since the method could be similarly run on other connected components) and the statistics of the largest connected components of networks are shown in Table 1.

In order to show the efficiency and effectiveness of our algorithms comprehensively, we use several types of labels to evaluate our algorithms. In Facebook and Google+, we use users' genders as node labels. In Pokec, we use users' locations as node labels. Such information can be obtained in the users' profiles in these networks. While in Orkut and Livejournal, the node degree is considered as the node label since we do not have the users' profiles in these two networks. Node degree contains structural information about the graph and in OSNs, it shows the number of friends that the user has. In order to simplify the discussion, all the labels are denoted by integers in the experiments.

Mixing Time. The mixing time of the Markov Chain is defined as the minimal length of the random walk in order to reach the stationary distribution. Following [2, 19], we define the mixing time of a Markov chain on G parameterized by a variation distance parameter ϵ as follows,

Definition 5.1. The mixing time parameterized by ϵ of a Markov Chain is defined as

$$T(\epsilon) = \max_i \min\{t : |\pi - \pi^{(i)}|_1 < \epsilon\} \\ = \max_i \min\{t : \frac{1}{2} \sum_{u \in V} |\pi(u) - [\pi^{(i)} P^t](u)| < \epsilon\} \quad (23)$$

where vector π is the stationary distribution and P is the transition matrix. Vector $\pi^{(i)}$ is the initial distribution concentrated at node i , i.e. the i -th element is 1 and all the other elements are 0. $[\pi^{(i)} P^t](u)$ is the u -th element in $\pi^{(i)} P^t$. $|\pi - \pi^{(i)}|_1$ is the total variation distance which is a distance measure of two probability distributions.

After testing, we find that when $\epsilon = 10^{-3}$ which is small enough, the mixing time of Facebook, Google+, Pokec, Orkut and Livejournal is 3200, 200, 100, 800 and 900 respectively which are not very large. So it is easy to achieve the stationary distribution quickly in our experiments. Note that the nodes or edges encountered in the random walk before the mixing time are not included in the sample set.

Table 1: Statistics of Datasets

Network	$ V $	$ E $
Facebook [15]	4.0×10^3	8.82×10^4
Google+ [15]	1.08×10^5	1.22×10^7
Pokec [15]	1.6×10^6	2.23×10^7
Orkut[1]	3.08×10^6	1.17×10^8
Livejournal[1]	4.8×10^6	4.28×10^7

Table 2: Abbreviations of Algorithms

Algorithm Name	Abbreviation
NeighborSample with the Hansen-Hurwitz estimator	NeighborSample-HH
NeighborSample with the Horvitz-Thompson estimator	NeighborSample-HT
NeighborExploration with the Hansen-Hurwitz estimator	NeighborExploration-HH
NeighborExploration with the Horvitz-Thompson estimator	NeighborExploration-HT
NeighborExploration with the with Re-weighted method	NeighborExploration-RW
Existing algorithm using re-weighted method	EX-RW
Existing algorithm using Metropolis-Hastings random walk	EX-MHRW
Existing algorithm using maximum degree random walk	EX-MD
Existing algorithm using Rejection-controlled Metropolis-Hastings Random Walk Algorithm on Edges	EX-RCMH
Existing algorithm using General Maximum Degree Random Walk Algorithm on Edges	EX-GMD

Adaptations of Existing Algorithms. In addition to the two algorithms and their five corresponding estimators developed in this paper, we consider a few baseline methods adapted from an existing study [16]. In [16], the authors have summarized several common used algorithms which perform random walk on nodes to get unbiased estimation of the relative count of target nodes which has a particular degree. If we multiply this estimation by the total number of nodes, then we can obtain the estimation of the count of target nodes. Those existing methods cannot be applied directly to our problem, since our problem is to estimate the number of target edges instead of target nodes. However, we find that if we transform the original graph G into a new graph G' , then we can apply those existing algorithms in [16] on graph G' to get the estimation of the count of target edges in G . We first describe how to construct G' base on G .

Let $G = (V, E)$ be the given graph, we construct a new graph $G' = (H, R)$ based on G with the following properties,

- Each edge in G corresponds to a node in G' and all these nodes constitute the node set H in G' . Thus, we have $|H| = |E|$.
- Two nodes in H are connected by an edge in G' if and only if they share one common vertex of G and all these edges constitute R .

where V and E are node set and edge set in G , and H and R are node set and edge set in G' .

It is obvious that if we apply the existing algorithms in [16] on graph G' , then we can get the estimation of the count of target nodes in G' . Since each node in G' corresponds to an edge in G , counting the number of target edges in G is the same as counting the number of target nodes in G' .

In [16], three existing algorithms, Re-weighted method, Metropolis-Hastings Random Walk algorithm (MHRW), and Maximum Degree Random Walk algorithm (MDRW), are reviewed by the authors. Also, two new algorithms, Rejection-controlled Metropolis-Hastings Random Walk algorithm (RCMH) and General Maximum Degree Random Walk algorithm (GMD), are proposed by the authors. Two parameters, α and δ , are used to control the performance of RCMH and GMD, respectively. The authors suggested to set $\alpha \in [0, 0.3]$ and $\delta \in [0.3, 0.7]$, and in this paper, we adopt settings which give the best results.

The abbreviation of each tested algorithm is shown in Table 2, and all algorithms are implemented in C++, and we conducted experiments on a Linux machine with Intel 3.40GHz CPU.

Measurements. We adopt the *normalized root mean square error* (NRMSE) measure as our error measurement, which is defined as follows.

$$\text{NRMSE}(\hat{F}) = \frac{\sqrt{\mathbb{E}[(\hat{F}-F)^2]}}{F} = \frac{\sqrt{\text{Var}[\hat{F}] + (F - \mathbb{E}[\hat{F}])^2}}{F}, \quad (24)$$

Note that NRMSE captures both the variance and the bias of the estimator.

Objectives. The objectives of the experiments can be summarized as follows:

- (1) The diversity of the network types and corresponding label types serve to show that our methods sustain satisfactory performance across different domains.
- (2) The different types of labels in different networks have very different frequencies. This helps us to investigate the effect of target edge frequency on the accuracy of the estimation.
- (3) Another factor which can affect the accuracy is the sample size, we expect the accuracy to improve with more samples taken. Hence in our experiments, we vary the sample sizes and examine the impact.
- (4) A major objective is to compare our proposed methods with the baseline methods, which are outlined in Section 5.1. We aim to show that our proposed algorithms outperform these baseline methods.
- (5) Since we propose two algorithms, NeighborSample and NeighborExploration, it is of interest to compare the two and find out how they differ and how to choose between these algorithms depending on the given problem characteristics.

5.2 Comparison among algorithms with varying sample size

Firstly, we compare the estimation accuracies of different algorithms. We examine the NRMSE results of different algorithms

Table 3: The labels and their corresponding locations in Pokec

Label	Location
2	zilinsky kraj, kysucke nove mesto
13	zahranicie, zahranicie - australia
20	kosicky kraj, michalovce
24	trnavsky kraj, trnava
51	trnavsky kraj, skalica
86	bratislavsky kraj, bratislava - nove mesto
122	kosicky kraj, kosice - ostatne
135	banskobystricky kraj, dudince

while varying the sample size from $0.5\%|V|$ to $5\%|V|$. Each target edge label is represented in the form of (A, B) where A and B are two integers representing two node labels.

In Facebook and Google+, we use one target edge label (1, 2) (1 and 2 represent female and male respectively), while in Pokec, Orkut and Livejournal, we pick 4 different target edge labels to evaluate all algorithms. The results on Facebook are shown in Table 4. The results on Google+ are shown in Table 5. The results on Pokec are shown in Tables 6 - 9 (We use 4 target edge labels, (86,135), (2,51), (13,20), and (24,122). All these numbers represent locations using Slovak language, which are shown in Table 3). The results on Orkut are shown in Tables 10 - 13. The results on Livejournal are shown in Tables 14 - 17.

In these tables, each row shows the NRMSE of an algorithm with increasing sample size and each column shows the NRMSE of each algorithm for a fixed sample size. The target edge label, the count and the percentage count of the target edges are shown in the caption of each table. Each NRMSE value is calculated by averaging over 200 independent simulations.

In Pokec, Orkut and Livejournal, there are thousands of edge labels we can choose. We first order those edge labels in ascending order of the count of target edges and divide them into 4 parts with equal size, then we pick one target edge label from each part randomly. With this method, we can test our algorithms on both high frequency edge labels and low frequency edge labels.

Tables 18 - 22 show the bounds of number of samples needed to achieve an $(0.1, 0.1)$ -approximation based Theorem 4.1 - 4.5. However, from the experimental results in Tables 4 - 17, we find that the number of samples needed to achieve a good estimation is much less than the bound.

The best NRMSE results for each sample size are underlined and marked with bold font. The best NRMSE results and the corresponding algorithms are also summarized in Tables 23 - 26 when $5\%|V|$ API calls are used.

We summarize our findings as follows.

- (1) The best algorithm in each table is always one of our newly proposed algorithms (NeighborSample and NeighborExploration), demonstrating that our new algorithms outperform adaptations of existing algorithms.
- (2) Our algorithms give good estimation with low API cost. Tables 23 - 26 summarize the best algorithms and the corresponding NRMSE values of each tested label when only $5\%|V|$ API calls are used. The largest NRMSE is 0.209 and most of the NRMSE values are smaller than 0.1. Note that for some tested target labels, the number of target edges is relatively small compared with the total number of edges, while our algorithms can still obtain accurate estimations. This shows that our proposed algorithms are highly effective.
- (3) The NRMSE results of all algorithms decrease as the number of API calls increases, which means that our estimation

Table 4: Facebook, target label=(1,2), number of target edges=37400, percentage=42.4%

	0.5% V	1.0% V	1.5% V	2.0% V	2.5% V	3.0% V	3.5% V	4.0% V	4.5% V	5.0% V
NeighborSample-HH	0.341	0.227	0.187	0.182	0.171	0.164	0.153	0.142	0.129	0.127
NeighborSample-HT	0.222	0.162	0.159	0.153	0.134	0.118	0.125	0.105	0.102	0.104
NeighborExploration-HH	0.284	0.334	0.247	0.29	0.272	0.164	0.21	0.234	0.178	0.186
NeighborExploration-HT	0.465	0.509	0.52	0.371	0.332	0.296	0.338	0.234	0.324	0.271
NeighborExploration-RW	3.881	2.919	3.857	2.781	2.482	2.891	1.584	2.279	2.363	2.339
EX-MDRW	0.875	0.741	0.676	0.692	0.575	0.554	0.559	0.531	0.485	0.456
EX-MHRW	0.377	0.299	0.246	0.245	0.241	0.182	0.183	0.19	0.164	0.157
EX-RW	0.338	0.244	0.219	0.215	0.177	0.17	0.193	0.148	0.157	0.172
EX-RCMH	0.645	0.513	0.437	0.387	0.421	0.386	0.298	0.30	0.321	0.318
EX-GMD	0.277	0.240	0.181	0.188	0.162	0.179	0.171	0.156	0.156	0.145

Table 5: Google+, target label=(1,2), number of target edges=328000, percentage=26.89%

	0.5% V	1.0% V	1.5% V	2.0% V	2.5% V	3.0% V	3.5% V	4.0% V	4.5% V	5.0% V
NeighborSample-HH	0.089	0.061	0.053	0.046	0.043	0.037	0.032	0.031	0.031	0.029
NeighborSample-HT	0.092	0.073	0.059	0.048	0.04	0.036	0.033	0.034	0.029	0.03
NeighborExploration-HH	0.7	0.689	0.642	0.627	0.647	0.58	0.558	0.582	0.49	0.491
NeighborExploration-HT	0.611	0.676	0.607	0.713	0.536	0.578	0.547	0.477	0.436	0.499
NeighborExploration-RW	13.506	11.856	16.765	21.985	19.323	16.279	15.079	11.97	6.65	16.06
EX-MDRW	0.478	0.451	0.443	0.379	0.24	0.269	0.259	0.225	0.261	0.207
EX-MHRW	0.169	0.118	0.089	0.078	0.075	0.06	0.066	0.053	0.057	0.055
EX-RW	0.162	0.117	0.113	0.08	0.078	0.07	0.066	0.067	0.058	0.051
EX-RCMH	0.161	0.108	0.09	0.074	0.066	0.051	0.062	0.063	0.052	0.043
EX-GMD	0.388	0.302	0.228	0.252	0.211	0.187	0.163	0.178	0.169	0.161

Table 6: Pokec, target label=(86,135), number of target edges=295, percentage=0.001%

	0.5% V	1.0% V	1.5% V	2.0% V	2.5% V	3.0% V	3.5% V	4.0% V	4.5% V	5.0% V
NeighborSample-HH	2.526	1.935	1.413	1.608	1.273	1.38	1.381	1.074	1.007	1.016
NeighborSample-HT	2.802	1.862	1.478	1.378	0.965	1.124	1.174	0.826	1.323	0.853
NeighborExploration-HH	0.761	0.606	0.445	0.339	0.386	0.426	0.302	0.36	0.238	0.209
NeighborExploration-HT	2.023	0.778	0.541	0.659	0.512	0.307	0.364	0.233	0.3	0.241
NeighborExploration-RW	1.861	0.685	0.542	0.466	0.325	0.362	0.457	0.317	0.355	0.307
EX-MDRW	1.0	1.0	1.0	1.0	104.73	1.0	16.607	2.222	13.005	1.0
EX-MHRW	3.492	2.597	1.935	1.783	1.184	1.521	1.527	2.105	1.136	1.47
EX-RW	3.52	2.22	2.656	2.555	1.472	1.533	1.292	1.532	1.237	1.921
EX-RCMH	0.949	0.607	0.450	0.477	0.430	0.405	0.303	0.352	0.314	0.226
EX-GMD	1.0	1.0	1.23	1.35	0.98	2.45	1.23	0.88	0.93	1.06

Table 7: Pokec, target label=(2,51),number of target edges=1163, percentage=0.005%

	0.5% V	1.0% V	1.5% V	2.0% V	2.5% V	3.0% V	3.5% V	4.0% V	4.5% V	5.0% V
NeighborSample-HH	1.262	1.036	0.748	0.73	0.701	0.649	0.551	0.478	0.503	0.444
NeighborSample-HT	1.62	1.17	0.768	0.742	0.739	0.578	0.559	0.599	0.599	0.461
NeighborExploration-HH	0.448	0.301	0.319	0.203	0.177	0.169	0.139	0.129	0.16	0.124
NeighborExploration-HT	0.424	0.401	0.235	0.203	0.196	0.159	0.188	0.139	0.149	0.149
NeighborExploration-RW	0.941	0.407	0.257	0.231	0.22	0.175	0.188	0.156	0.172	0.155
EX-MDRW	1.0	3.104	13.812	1.0	27.873	2.122	1.649	4.274	2.599	2.392
EX-MHRW	1.494	1.248	1.132	0.886	0.987	0.759	0.611	0.628	0.506	0.624
EX-RW	1.905	1.604	1.4	0.996	0.921	0.665	0.719	0.751	0.655	0.528
EX-RCMH	1.65	1.00	0.971	0.759	0.648	0.709	0.628	0.511	0.613	0.497
EX-GMD	1.0	5.79	1.0	1.07	1.57	6.08	1.34	3.36	1.68	1.25

Table 8: Pokec, target label=(13,20), number of target edges=2134, percentage=0.01%

	0.5% V	1.0% V	1.5% V	2.0% V	2.5% V	3.0% V	3.5% V	4.0% V	4.5% V	5.0% V
NeighborSample-HH	1.108	0.85	0.635	0.696	0.522	0.531	0.448	0.374	0.404	0.36
NeighborSample-HT	1.555	0.877	0.72	0.552	0.565	0.607	0.458	0.381	0.406	0.382
NeighborExploration-HH	0.396	0.264	0.228	0.192	0.164	0.176	0.139	0.137	0.136	0.12
NeighborExploration-HT	0.445	0.28	0.205	0.211	0.173	0.156	0.15	0.128	0.138	0.104
NeighborExploration-RW	0.344	0.275	0.214	0.194	0.149	0.163	0.144	0.135	0.127	0.146
EX-MDRW	7.56	1.0	9.953	11.815	25.159	3.314	8.077	8.987	3.582	2.476
EX-MHRW	1.373	1.291	0.935	0.706	0.695	0.552	0.545	0.546	0.539	0.415
EX-RW	1.803	1.885	0.864	0.679	0.678	0.58	0.616	0.639	0.451	0.548
EX-RCMH	1.21	0.811	0.625	0.877	0.541	0.461	0.496	0.442	0.527	0.419
EX-GMD	1.27	1.0	1.0	1.28	1.00	1.77	3.05	2.30	2.67	1.24

Table 9: Pokec, target label=(24,122), number of target edges=5784, percentage=0.03%

	0.5% V	1.0% V	1.5% V	2.0% V	2.5% V	3.0% V	3.5% V	4.0% V	4.5% V	5.0% V
NeighborSample-HH	0.727	0.532	0.41	0.358	0.291	0.314	0.282	0.25	0.229	0.213
NeighborSample-HT	0.839	0.46	0.409	0.292	0.226	0.34	0.25	0.292	0.267	0.189
NeighborExploration-HH	0.349	0.247	0.196	0.192	0.154	0.124	0.141	0.115	0.096	0.101
NeighborExploration-HT	0.382	0.29	0.214	0.156	0.178	0.143	0.117	0.118	0.107	0.093
NeighborExploration-RW	0.342	0.251	0.204	0.214	0.165	0.147	0.122	0.115	0.121	0.095
EX-MDRW	1.163	20.821	6.422	2.987	2.546	4.971	2.431	6.339	2.183	5.172
EX-MHRW	1.063	0.592	0.516	0.57	0.452	0.354	0.387	0.324	0.393	0.294
EX-RW	0.996	0.84	0.643	0.455	0.482	0.467	0.501	0.39	0.328	0.334
EX-RCMH	0.949	0.607	0.450	0.477	0.430	0.405	0.303	0.352	0.314	0.226
EX-GMD	1.67	1.12	3.03	2.58	2.30	1.81	1.41	1.09	1.25	1.90

Table 10: Orkut, target label=(48,45), number of target edges=5627, percentage=0.001%

	0.5% V	1.0% V	1.5% V	2.0% V	2.5% V	3.0% V	3.5% V	4.0% V	4.5% V	5.0% V
NeighborSample-HH	1.08	0.884	0.688	0.705	0.578	0.473	0.402	0.379	0.436	0.332
NeighborSample-HT	0.917	0.812	0.487	0.687	0.689	0.343	0.485	0.533	0.341	0.332
NeighborExploration-HH	0.315	0.265	0.195	0.172	0.146	0.141	0.099	0.116	0.096	0.089
NeighborExploration-HT	0.395	0.237	0.154	0.185	0.14	0.117	0.123	0.105	0.109	0.114
NeighborExploration-RW	0.479	0.304	0.215	0.185	0.161	0.156	0.124	0.118	0.116	0.099
EX-MDRW	1.407	9.96	12.876	13.999	10.188	4.846	4.834	3.759	2.085	3.039
EX-MHRW	1.51	0.852	0.843	0.673	0.601	0.613	0.505	0.471	0.429	0.41
EX-RW	1.181	0.693	0.599	0.558	0.542	0.512	0.378	0.41	0.366	0.373
EX-RCMH	0.944	0.670	0.649	0.524	0.425	0.362	0.37	0.379	0.35	0.32
EX-GMD	1.0	1.34	3.41	1.48	1.28	1.40	1.50	1.70	1.35	1.44

Table 11: Orkut, target label=(11,0),number of target edges=49879, percentage=0.043%

	0.5% V	1.0% V	1.5% V	2.0% V	2.5% V	3.0% V	3.5% V	4.0% V	4.5% V	5.0% V
NeighborSample-HH	0.386	0.357	0.231	0.237	0.168	0.186	0.163	0.148	0.147	0.142
NeighborSample-HT	0.284	0.278	0.249	0.268	0.216	0.197	0.149	0.136	0.167	0.152
NeighborExploration-HH	0.491	0.331	0.278	0.228	0.207	0.193	0.168	0.143	0.147	0.147
NeighborExploration-HT	0.425	0.286	0.274	0.21	0.198	0.185	0.156	0.143	0.152	0.122
NeighborExploration-RW	0.31	0.268	0.202	0.188	0.151	0.166	0.149	0.122	0.111	0.124
EX-MDRW	8.977	6.288	2.317	6.809	8.318	7.408	8.366	3.708	2.174	2.973
EX-MHRW	0.662	0.49	0.381	0.368	0.345	0.261	0.291	0.243	0.238	0.212
EX-RW	1.005	0.788	0.678	0.544	0.503	0.424	0.37	0.379	0.355	0.373
EX-RCMH	0.997	0.651	0.491	0.453	0.384	0.312	0.268	0.281	0.271	0.261
EX-GMD	0.995	3.51	1.78	3.11	1.66	1.76	1.39	2.35	1.71	1.47

Table 12: Orkut, target label=(1,0),number of target edges=128501, percentage=0.11%

	0.5% V	1.0% V	1.5% V	2.0% V	2.5% V	3.0% V	3.5% V	4.0% V	4.5% V	5.0% V
NeighborSample-HH	0.198	0.162	0.124	0.112	0.113	0.1	0.086	0.081	0.075	0.068
NeighborSample-HT	0.182	0.125	0.106	0.084	0.088	0.085	0.067	0.064	0.065	0.063
NeighborExploration-HH	0.491	0.331	0.278	0.228	0.207	0.193	0.168	0.143	0.147	0.147
NeighborExploration-HT	0.212	0.156	0.136	0.113	0.11	0.087	0.089	0.077	0.063	0.071
NeighborExploration-RW	0.523	0.35	0.253	0.215	0.189	0.187	0.17	0.136	0.154	0.15
EX-MDRW	8.977	6.288	2.317	6.809	8.318	7.408	8.366	3.708	2.174	2.973
EX-MHRW	11.117	27.794	11.387	8.223	3.273	4.57	1.0	1.619	4.748	5.788
EX-RW	0.662	0.49	0.381	0.368	0.345	0.261	0.291	0.243	0.238	0.212
EX-RCMH	1.18	0.985	0.75	0.688	0.511	0.553	0.473	0.436	0.427	0.468
EX-GMD	1.0	5.72	5.74	1.22	3.26	1.97	1.45	2.05	1.69	2.92

Table 13: Orkut, target label=(6,5),number of target edges=769188, percentage=0.657%

	0.5% V	1.0% V	1.5% V	2.0% V	2.5% V	3.0% V	3.5% V	4.0% V	4.5% V	5.0% V
NeighborSample-HH	0.124	0.079	0.067	0.066	0.056	0.054	0.043	0.045	0.042	0.038
NeighborSample-HT	0.107	0.072	0.071	0.057	0.04	0.04	0.049	0.043	0.032	0.039
NeighborExploration-HH	0.159	0.105	0.096	0.077	0.075	0.067	0.07	0.053	0.053	0.05
NeighborExploration-HT	0.136	0.105	0.1	0.085	0.07	0.063	0.056	0.06	0.054	0.046
NeighborExploration-RW	0.084	0.063	0.057	0.043	0.045	0.04	0.037	0.029	0.028	0.029
EX-MDRW	3.514	2.366	2.551	2.373	1.451	1.403	1.6	1.29	1.297	1.026
EX-MHRW	0.186	0.128	0.105	0.099	0.091	0.082	0.074	0.062	0.068	0.055
EX-RW	0.352	0.231	0.214	0.156	0.153	0.133	0.115	0.116	0.102	0.096
EX-RCMH	0.238	0.178	0.159	0.116	0.116	0.091	0.101	0.087	0.082	0.078
EX-GMD	1.84	0.914	0.904	0.712	0.77	0.705	0.646	0.683	0.693	0.64

Table 14: Livejournal, target label=(34,12),number of target edges=5168, percentage=0.001%

	0.5% V	1.0% V	1.5% V	2.0% V	2.5% V	3.0% V	3.5% V	4.0% V	4.5% V	5.0% V
NeighborSample-HH	0.62	0.445	0.338	0.308	0.25	0.232	0.272	0.254	0.179	0.198
NeighborSample-HT	0.6	0.45	0.259	0.252	0.326	0.243	0.175	0.24	0.209	0.218
NeighborExploration-HH	0.264	0.164	0.158	0.14	0.138	0.094	0.085	0.098	0.089	0.088
NeighborExploration-HT	0.231	0.168	0.173	0.114	0.2	0.117	0.144	0.083	0.086	0.074
NeighborExploration-RW	1.089	0.205	0.244	0.167	0.121	0.108	0.112	0.113	0.099	0.091
EX-MDRW	3.482	1.666	3.297	3.952	2.436	4.498	2.553	2.273	1.647	3.465
EX-MHRW	0.669	0.589	0.422	0.373	0.318	0.303	0.278	0.28	0.247	0.245
EX-RW	0.587	0.451	0.324	0.382	0.267	0.253	0.19	0.161	0.213	0.179
EX-RCMH	0.564	0.343	0.303	0.263	0.248	0.216	0.199	0.186	0.171	0.159
EX-GMD	1.86	1.72	1.70	0.991	1.60	1.30	1.00	0.850	1.11	0.987

Table 15: Livejournal, target label=(19,16), number of target edges=15442, percentage=0.04%

	0.5% V	1.0% V	1.5% V	2.0% V	2.5% V	3.0% V	3.5% V	4.0% V	4.5% V	5.0% V
NeighborSample-HH	0.442	0.291	0.196	0.198	0.193	0.174	0.139	0.144	0.116	0.119
NeighborSample-HT	0.557	0.172	0.257	0.182	0.16	0.146	0.157	0.166	0.129	0.117
NeighborExploration-HH	0.393	0.277	0.265	0.204	0.15	0.136	0.125	0.136	0.118	0.105
NeighborExploration-HT	0.466	0.293	0.236	0.204	0.156	0.164	0.157	0.132	0.133	0.115
NeighborExploration-RW	0.543	0.327	0.278	0.263	0.167	0.159	0.173	0.154	0.13	0.129
EX-MDRW	3.447	4.478	2.861	1.834	2.201	1.527	4.163	1.615	1.89	2.148
EX-MHRW	0.637	0.356	0.303	0.32	0.242	0.233	0.233	0.221	0.187	0.187
EX-RW	0.742	0.359	0.337	0.275	0.333	0.241	0.198	0.194	0.172	0.167
EX-RCMH	0.476	0.282	0.239	0.257	0.224	0.195	0.157	0.151	0.128	0.138
EX-GMD	2.52	1.30	1.26	1.23	1.16	1.33	0.853	0.980	0.735	0.822

Table 16: Livejournal, target label=(8,4), number of target edges=203945 percentage=0.48%

	0.5% V	1.0% V	1.5% V	2.0% V	2.5% V	3.0% V	3.5% V	4.0% V	4.5% V	5.0% V
NeighborSample-HH	0.104	0.092	0.082	0.06	0.051	0.048	0.038	0.04	0.039	0.04
NeighborSample-HT	0.105	0.101	0.08	0.053	0.06	0.042	0.048	0.046	0.04	0.04
NeighborExploration-HH	0.138	0.107	0.09	0.078	0.057	0.055	0.067	0.06	0.043	0.048
NeighborExploration-HT	0.135	0.117	0.101	0.103	0.082	0.095	0.105	0.084	0.094	0.087
NeighborExploration-RW	0.152	0.1	0.068	0.061	0.084	0.056	0.055	0.054	0.061	0.039
EX-MDRW	1.761	1.535	1.613	1.191	1.191	1.066	1.442	0.942	1.03	0.781
EX-MHRW	0.19	0.127	0.112	0.094	0.071	0.073	0.064	0.064	0.047	0.051
EX-RW	0.201	0.167	0.134	0.096	0.104	0.084	0.08	0.083	0.074	0.082
EX-RCMH	0.172	0.128	0.105	0.093	0.073	0.0745635	0.07	0.08	0.056	0.053
EX-GMD	1.32	0.835	0.717	0.666	0.642	0.613	0.591	0.592	0.546	0.541

Table 17: Livejournal, target label=(1,0), number of target label=1753000, percentage=4.1%

	0.5% V	1.0% V	1.5% V	2.0% V	2.5% V	3.0% V	3.5% V	4.0% V	4.5% V	5.0% V
NeighborSample-HH	0.094	0.054	0.053	0.041	0.04	0.041	0.031	0.029	0.026	0.028
NeighborSample-HT	0.07	0.057	0.048	0.04	0.037	0.031	0.035	0.027	0.027	0.025
NeighborExploration-HH	0.152	0.083	0.072	0.051	0.052	0.049	0.042	0.037	0.035	0.033
NeighborExploration-HT	0.119	0.089	0.076	0.067	0.056	0.05	0.049	0.044	0.042	0.044
NeighborExploration-RW	0.053	0.048	0.043	0.033	0.031	0.026	0.027	0.024	0.023	0.02
EX-MDRW	3.332	1.757	1.825	0.978	1.14	1.213	1.101	0.95	0.884	0.935
EX-MHRW	0.196	0.131	0.09	0.096	0.079	0.07	0.079	0.068	0.057	0.064
EX-RW	0.252	0.211	0.186	0.145	0.122	0.107	0.11	0.128	0.103	0.08
EX-RCMH	0.155	0.175	0.133	0.111	0.0786	0.092	0.08	0.09	0.067	0.073
EX-GMD	1.16	0.958	0.867	0.726	0.705	0.654	0.642	0.647	0.686	0.59

Table 18: Bound on the number of samples in Facebook

	NeighborSample-HH	NeighborSample-HT	NeighborExploration-HH	NeighborExploration-HT	NeighborExploration-RW
(1,2)	1359	5398	921	3151	53427

Table 19: Bound on the number of samples in Google+

	NeighborSample-HH	NeighborSample-HT	NeighborExploration-HH	NeighborExploration-HT	NeighborExploration-RW
(1,2)	2726	13879	1714	25400	445515

converges to the ground truth when more samples or API calls are used. This behavior is as expected.

- (4) In most of the cases, NeighborExploration returns the best estimations. However, NeighborSample outperforms NeighborExploration in the cases where the target edges constitute a larger proportion in the whole edge set. This is the case for the results of facebook and google+. This indicates that when the target edges are abundant, neighborhood exploration is not needed to boost the sampling probability for the target edges.
- (5) For the datasets of Orkut and Livejournal, we have multiple sets of results for edge labels with different frequencies. It is found that the NRMSE values for more frequent labels are generally smaller than those with less frequent labels. We have therefore conducted a more systematic study about the impact of the label frequency, to be reported in the next subsection.

5.3 Comparison among algorithms with varying relative count of target edges

We notice that in the same graph, for different target labels the best algorithms can be different. It turns out that the relative count of target edges ($F/|E|$) may also affect the performance of different algorithms. In order to study the relationship between the performance of different algorithms and the relative count of target edges, we measure the values of NRMSE for a range of $F/|E|$. The results for Orkut and Livejournal are plotted in Figure 1 and Figure 2. Each node in these figures corresponds a target edge label and the x-coordinate is the relative count of edges with this label and the y-coordinate is the NRMSE of the target edge count estimation when $5\%|V|$ API calls are used. Here, we only run experiments on two networks, Orkut and Livejournal, since the range of the relative count of target edges in Orkut and Livejournal is much larger than the ranges in other

networks, so the change of NRMSE is more obvious in these two networks when the relative count of target edges varies. The results of existing algorithms are not shown, since we have demonstrated that those algorithms are much less competitive in the previous experiments. Each NRMSE value is calculated by averaging over 200 independent simulations.

We summarize our results as follows.

- (1) In the same network, as the relative count of target edges increases, the NRMSE results of all algorithms decrease, which means that the estimation is more accurate. This is reasonable, since the probability of sampling target edges in random walk will be higher if there are more target edges in the networks, which will result in better estimations.
- (2) When the relative count of target edges changes the best algorithm may also be different. When the relative count of target edges is small, NeighborExploration algorithms outperforms NeighborSample algorithms and the difference is quite significant, but when the relative count of target edges is large enough, the results of NeighborExploration algorithms and NeighborSample algorithms are very close and the best algorithm will change from case to case.

We explain why our new algorithm NeighborExploration outperforms NeighborSample when the relative count of target edges is small as follows. In NeighborSample algorithms, the probability of sampling a target edge is $\frac{F}{|E|}$, since NeighborSample samples edges uniformly. However, our new algorithm, NeighborExploration, can find a target edge with probability $\frac{\sum_{u \in Q} d_u}{2|E|}$, where node set Q contains all nodes which is included in at least one target edges, since once we sample a node all target edges which contains this node will also be found. As a result, our new algorithm NeighborExploration can obtain target edges with a

Table 20: Bounds on the number of samples in Pokec

	NeighborSample-HH	NeighborSample-HT	NeighborExploration-HH	NeighborExploration-HT	NeighborExploration-RW
(86,135)	7.56×10^7	2.77×10^8	4.08×10^6	3.77×10^8	7.35×10^7
(2,51)	1.91×10^7	2.16×10^8	6.9×10^5	2.54×10^8	1.25×10^7
(13,20)	1.04×10^7	1.89×10^8	4.6×10^5	2.01×10^8	8.27×10^6
(24,122)	3.85×10^6	1.45×10^8	2.3×10^5	1.15×10^8	4.16×10^6

Table 21: Bounds on the number of samples in Orkut

	NeighborSample-HH	NeighborSample-HT	NeighborExploration-HH	NeighborExploration-HT	NeighborExploration-RW
(48,45)	2.08×10^7	9.62×10^8	2.46×10^5	4.8×10^6	4.44×10^6
(11,0)	2.34×10^6	4.5×10^8	3.3×10^5	1.03×10^8	6.03×10^6
(1,0)	9.1×10^5	2.45×10^8	1.8×10^5	3.97×10^7	3.34×10^6
(6,5)	1.5×10^5	2.11×10^7	1.3×10^4	1.38×10^6	2.49×10^5

Table 22: Bounds on the number of samples in Livejournal

	NeighborSample-HH	NeighborSample-HT	NeighborExploration-HH	NeighborExploration-HT	NeighborExploration-RW
(34,12)	8.28×10^6	3.16×10^8	1.8×10^5	5.86×10^6	3.23×10^6
(19,16)	2.77×10^6	2.22×10^8	9.6×10^4	4.45×10^6	1.74×10^6
(8,4)	2.09×10^5	3.0×10^7	1.7×10^4	7.10×10^6	3.09×10^5
(6,5)	2.34×10^4	5.93×10^5	9.8×10^3	6.0×10^5	1.76×10^5

higher probability than NeighborSample, especially when $F/|E|$ is small, so NeighborExploration performs better.

Table 23: Best algorithm for Facebook and Google+ using 5%|V| API calls

Social Network	Label	Best algorithm	NRMSE
Facebook	(1,2)	NeighborSample-HT	0.104
Google+	(1,2)	NeighborSample-HH	0.029

Table 24: Best algorithm for Pokec using 5%|V| API calls

Label	Best algorithm	NRMSE
(135,86)	NeighborExploration-HH	0.209
(2,51)	NeighborExploration-HH	0.124
(13,20)	NeighborExploration-HH	0.12
(24,122)	NeighborExploration-HT	0.093

Table 25: Best algorithm for Orkut using 5%|V| API calls

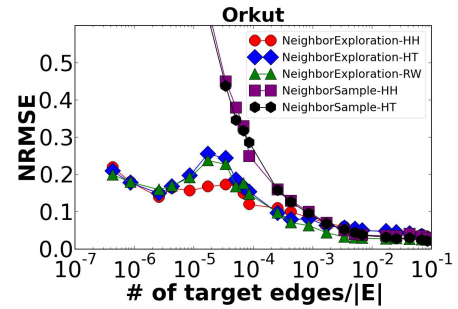
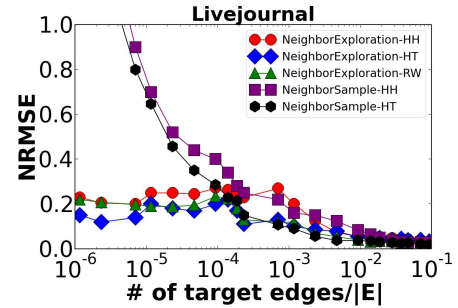
Label	Best algorithm	NRMSE
(48,45)	NeighborExploration-HH	0.089
(11,0)	NeighborExploration-RW	0.124
(1,0)	NeighborSample-HT	0.063
(6,5)	NeighborSample-RW	0.029

Table 26: Best algorithm for Livejournal using 5%|V| API calls

Label	Best algorithm	NRMSE
(34,12)	NeighborExploration-HT	0.074
(19,16)	NeighborExploration-HH	0.105
(8,4)	NeighborExploration-RW	0.039
(1,0)	NeighborExploration-RW	0.02

6 CONCLUSION

In this paper, we propose to estimate the number of edges with target labels, which to the best of our knowledge corresponds to the first attempt to estimate graph properties refined by users' labels. To solve the problem, we developed two algorithms, namely NeighborSample and NeighborExploration, which samples a set of edges/nodes first and then constructs estimators based on the sampled edges/nodes. These two algorithms are suitable in different cases, e.g., NeighborExploration is better than NeighborSample when the fraction of edges with target labels is low. We also provide some theoretical results on the accuracy guarantees of the algorithms. We conducted extensive experiments which

**Figure 1: NRMSE vs. number of target edges in Orkut when 5%|V| API calls are used****Figure 2: NRMSE vs. number of target edges in Livejournal when 5%|V| API calls are used**

verified that the algorithms developed in this paper are superior over baseline methods.

There are a few directions for future study. For example, it would be interesting to estimate some other types of graph properties such as numbers of wedges and triangles refined by users' labels in OSNs.

REFERENCES

- [1] 2016. KONECT Datasets: The koblenz network collection. <http://konect.uni-koblenz.de>. (2016).
- [2] Louigi Addario-Berry and Tao Lei. 2015. The Mixing Time of the Newman-Watts Small-World Model. *Advances in Applied Probability* 47, 1 (2015), 37–56.
- [3] P. Anchuri, M.J. Zaki, O. Barkol, S. Golan, and M. Shamy. 2013. Approximate Graph Mining with Label Costs. *In KDD* (2013), 518–526.
- [4] A. Arora, M. Sachan, and A. Bhattacharya. 2014. Mining Statistically Significant Connected Subgraphs in Vertex Labeled Graphs. *In SIGMOD* (2014), 1003–1014.
- [5] X. Chen, Y. Li, G. P. Wang, and J. C. S. Lui. 2016. A general framework for estimating graphlet statistics via random walk. *Proc. VLDB Endow.* 10, 3 (2016), 253–264.
- [6] Y. Press D. A. Levin and E. L. Wilmer. 2008. *Markov Chains and Mixing Times*. American Mathematical Society.

- [7] M. Gjoka, M. Kurant, and C. T. Butts. 2010. Walking in Facebook: A Case Study of Unbiased Sampling of OSNs. *In INFOCOM (2010)*, 1–9.
- [8] Olle Häggström. 2002. *Finite Markov chains and algorithmic applications*. Vol. 52. Cambridge University Press.
- [9] J. Han and J.-R. Wen. 2013. Mining Frequent Neighborhood Patterns in a Large Labeled Graph. *In CIKM (2013)*, 259–268.
- [10] M. Hansen and W. Hurwitz. 1943. On the Theory of Sampling from Finite Populations. *In Annals of Mathematical Statistics* 14, 4 (1943), 333–362.
- [11] S. J. Hardiman and L. Katzir. 2013. Estimating clustering coefficients and size of social networks via random walk. *In WWW (2013)*, 539–550.
- [12] D. G. Horvitz and D. J. Thompson. 1952. A Generalization of Sampling Without Replacement from a Finite Universe. *J. Amer. Statist. Assoc.* 47, 260 (1952), 663–685.
- [13] L. Katzir, E. Liberty, and O. Somekh. 2011. Estimating sizes of social networks via biased sampling. *In WWW (2011)*, 597–606.
- [14] C.-H. Lee, X. Xu, and D. Y. Eun. 2012. Beyond random walk and metropolis-hastings samplers: why you should not backtrack for unbiased graph sampling. *In SIGMETRICS (2012)*, 319–330.
- [15] J. Leskovec and A. Krevl. 2016. SNAP Datasets: Stanford large network dataset collection. <http://snap.stanford.edu/data>. (2016).
- [16] R.-H. Li, J. Yu, L. Qin, R. Mao, and T. Jin. 2015. On random walk based graph sampling. *In ICDE (2015)*, 927–938.
- [17] J. S. Liu. 2001. Monte Carlo Strategies in Scientific Computing. *Springer* (2001).
- [18] László Lovász. 1993. Random Walks on Graphs: A Survey. *In Combinatorics, Paul Erdos is Eighty 2* (1993).
- [19] Abedelaziz Mohaisen, Aaram Yun, and Yongdae Kim. 2010. Measuring the mixing time of social graphs. *In Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. ACM, 383–389.
- [20] S. Nandanwar and M.N. Murty. 2016. Structural Neighborhood Based Classification of Nodes in a Network. *In KDD (2016)*, 518–526.
- [21] P. Wang, J. C. S. Lui, B. Ribeiro, D. Towsley, J. Zhao, and X. Guan. 2014. Efficiently estimating motif statistics of large networks. *In ICDE (2014)*, 8:1–8:27.
- [22] P. Wang, J. Zhao, J. C. Lui, D. Towsley, and X. Guan. 2015. Unbiased characterization of node pairs over large graphs. *In TKDD* 9, 3 (2015).
- [23] Y. Wu, C. Long, A. W. Fu, and Z. Chen. 2017. Counting Edges and Triangles in Online Social Networks via Random Walk. *In APWeb-WAIM (2017)*, 346–361.
- [24] W. Ye, L. Zhou, D. Mautz, C. Plant, and C. Böhm. 2017. Learning from Labeled and Unlabeled Vertices in Networks. *In KDD (2017)*, 1265–1274.

A CHEBYSHEV’S INEQUALITY

Let X be a random variable with expectation $E[X]$ and variance $\text{var}(X)$. Then the Chebyshev’s inequality states that for any $t > 0$,

$$P(|X - E[X]| > t) \leq \frac{\text{var}(X)}{t^2} \quad (25)$$

Let \hat{X} be an estimator of $E[X]$, $t = \epsilon E[X]$ where $0 < \epsilon < 1$ and $0 < \delta < 1$, then we call \hat{X} an (ϵ, δ) -approximation of $E[X]$, if the following Chebyshev’s inequality holds.

$$P(|\hat{X} - E[X]| > \epsilon E[X]) \leq \frac{\text{var}(X)}{(\epsilon E[X])^2} \leq \delta \quad (26)$$

which shows that the probability of that \hat{X} is in the range $[(1 - \epsilon)E[X], (1 + \epsilon)E[X]]$ is larger than $1 - \delta$.

An Homophily-based Approach for Fast Post Recommendation on Twitter

Quentin Grossetti
CNAM, UPMC
Paris, France
quentin.grossetti@upmc.fr

Camelia Constantin
University Pierre et Marie Curie
Paris, France
camelia.constantin@lip6.fr

Cédric du Mouza
CNAM
Paris, France
dumouza@cnam.fr

Nicolas Travers
CNAM
Paris, France
nicolas.travers@cnam.fr

ABSTRACT

With the unprecedented growth of user-generated content produced on microblogging platforms, finding interesting content for a given user has become a major issue. However due to the intrinsic properties of microblogging systems, such as the volumetry, the short lifetime of posts and the sparsity of interactions between users and content, recommender systems cannot rely on traditional methods, such as collaborative filtering matrix factorization. After a thorough study of a large Twitter dataset, we present a propagation model which relies on homophily to propose post recommendations. Our approach relies on the construction of a similarity graph based on retweet behaviors on top of the Twitter graph. Finally we conduct experiments on our real dataset to demonstrate the quality and scalability of our method. **Keywords:** Recommender System; Collaborative Filtering; Microblogging Systems

1 INTRODUCTION

During the last decade, several microblogging platforms have emerged such as Twitter, Pinterest, Instagram, Weibo or Tumblr. These platforms rely on the same paradigm: users follow each other's and share content. They have their specific audience and features but all have encountered unprecedented growth. This growth tends to make microblogging platforms overcrowded and users to encounter difficulties to keep up with all the available content. For instance, in 2017, 500 million messages were published every day on Twitter. Finding relevant messages to recommend in real time is a real challenge of prime interest for these platforms. Indeed, effectively recommending fresh publications leads to higher engagement from users which is crucial for a social media service [12].

Many recommendations methods have been proposed in the literature. Some works propose to extract features from items to recommend them to suitable users like [23]. However this approach provides poor results on microblogging platforms due to the short length of the messages (140 characters for a tweet), and a broad variety of content: video, picture, sound etc. Other methods based on collaborative filtering try to capture similarity between users based on "co-liked" items and to use this similarity to determine recommendations [30]. Among collaborative filtering methods, matrix factorization [20] and social trust models [25]

are very popular. If these methods provide relevant recommendations to end-users, they can not scale up with the huge amount of items produced by a platform like Twitter despite existing optimization techniques [26]. For instance the similarity matrix size will be of 1 000 billion of messages with millions of users, and will permanently grow up, so in addition to storage issues (even if this is a sparse matrix), it requires constant and costly computations.

To face this issue, Twitter has developed its own recommender system named *GraphJet* [32]. This method relies on a bipartite graph built on users interactions with messages and computes random walks. By starting these walks from a query user, it is possible to compute a list of personalized recommendation at low cost [32]. However, due to its random walk-based computations, *GraphJet* tends towards recommending mostly popular messages. Even if many users tends to focus on popular tweets, this approach reduces chances to recommend more specific content with similar interests in the neighborhood.

Based on a thorough analysis of a real Twitter dataset of 2.2M users extracted in 2015, with a special focus on the homophily concept, we propose in this article an original approach which achieves a good trade-off between relevance, scalability and freshness of recommendations. Our intuition is that recommendation relevance can be enhanced by understanding how people with similar profiles are interconnected. Our proposal relies on *SimGraph*, a similarity graph which links users with relevant-based edges based on common retweets, along with a scalable propagation model.

In a nutshell, the main contributions of this work are:

- (1) a micro-blogging analysis which focuses on retweets and homophily characterization;
- (2) a novel method to drastically reduce the cost of collaborative filtering methods relying on homophily and the construction of a similarity graph *SimGraph*
- (3) a convergent and scalable propagation algorithm enhancing recommendations with transitivity along with optimization techniques;
- (4) a thorough experimental comparison between *Collaborative Filtering*, *GraphJet*, a *Bayes Inference Model* and *SimGraph*, focusing on recommendation quality, processing time and robustness over time.

The rest of the paper is organized as follows. Section 2 is a review of the related works. Section 3 provides a thorough analysis of a Twitter dataset with a focus on homophily, which leads to our similarity graph and propagation model in Section 4. Our optimized propagation algorithm is presented in Section 5

and compared to other solutions in experiments in Section 6. Finally, Section 7 concludes the paper.

2 RELATED WORK

Since their appearance in the beginning of the 1990s at Xerox [11], recommender systems have been extremely popular and are used to recommend a large variety of objects such as : music [29], movies [27], news [9], recipes [8], etc. Different types of recommending methods emerged: content-based, collaborative filtering, graph-based, bipartite networks or deep learning.

Content-based methods aim at describing both profiles and items in order to provide recommendation [23]. The strength of content-based methods lies in the capacity to provide recommendations without requiring any feedback from users by extracting tendencies of collected data. In the context of microblogging, the content itself is poor with small-size tweets, links or multimedia, even if some works on Twitter targeted this approach [18]. However, content-based models generally suffer from overspecialized recommendations [1].

Collaborative Filtering (*CF*), on the opposite, combines both content and user interactions in order to produce recommendations. *CF* is generally represented as a matrix resolution problem and provides relevant recommendation. Lot of works focus on sparseness [19], on scalability [26] and on cold-start problems [2]. Matrix factorization methods transform both user and item vectors to a latent factor space, where similarities between items and users are generated by lower-dimensional hidden factors. However in the Twitter context, matrix factorization is not scalable due to the matrix size and continuous growth, but also it does not allow to consider the freshness of the recommended items.

Some other proposals exploit the social network and weight edges to quantify correlations between users. Jiang et al. [17] mix the social network with individual preferences into a matrix factorization model and improve recommendations. A broad range of social recommender systems have a similar approach like [33]. In Jamali et al. [16], authors present *SocialMF* to also enrich matrix factorization with the network of users. However the size of the Twitter matrix (more than 100 billions values) and its high growth (500 millions of messages/day) make these methods hard to exploit. Many works on trust networks have been done with explicit information, like in SoRec [24] on Epinion¹ producing and factorizing probabilistic matrices. However, the relationships between users on microblogging platforms is very heterogeneous and can carry many different meanings. Consequently it can not be really considered as a trust relationship.

The bipartite network model is another popular approach in recommendation systems. For instance Twitter proposes in [32] *GraphJet*, a random walk approach on user and post graph. Even though this method is particularly efficient, it only focuses, for scalability purposes, on the freshness of interactions. In our point of view, *GraphJet* by missing to exploit older interactions reduce the complexity of users profiles. Moreover, we will see in the experiments that this solution mainly addresses the hot-topic recommendations.

There exist also other works on Twitter for recommending hashtags [10] or users [6], or for filtering messages in the timeline [34]. But very few address the post recommendation problem.

Finally, deep learning methods on huge datasets such as Twitter [28] or Youtube [7] process neural networks to connect many dimensions such as similarity matrix, content features,

# nodes	2.2M
# edges	325.5M
# tweets	3,002M
avg. out-deg.	57.8
avg. in-deg.	69.4
max out-deg.	349K
max in-deg.	185K
diameter	15
avg. path length	3.7

Table 1: Main features of the Twitter dataset

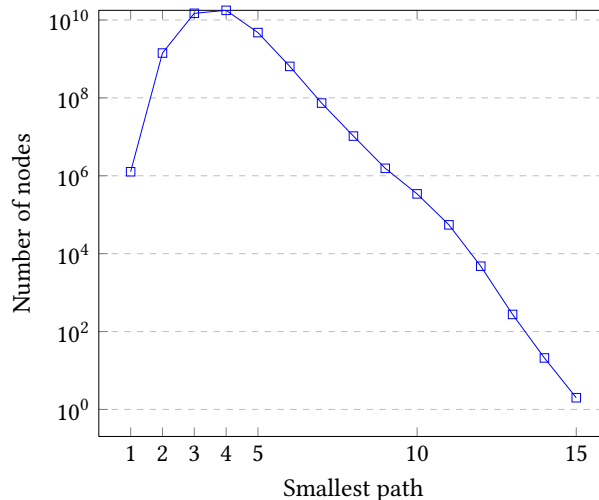


Figure 1: Twitter smallest paths distribution

user feedback. Even if the efficiency of such systems is promising, it faces scalability and dynamicity challenges for microblogging platforms. As a matter of facts, it currently uses past content to extract interesting items from user timelines but does not support real-time recommendation of posts from the whole network.

3 MICROBLOGGING ANALYSIS

We introduce here the main characteristics of our Twitter dataset and the different experiments we performed to characterize user behaviors. We discuss the main results of our data analysis and their implication for our recommendation model.

To build our dataset we first extracted a connected component from the graph from Kwak et al. [21]. Then for each node of this subgraph we updated every information in 2015 thanks to the Twitter API². More precisely, for each account we collected the incoming edges (followers), outgoing edges (followees) and all the tweets published by the associated account. Observe that due to the API limit we only retrieved the last 3,200 messages (maximum) for each account. Table 1 summarizes the main features of the dataset. With more than 2 million users and 3 billion messages we have a mean number of 1,375 published tweets per user. The original Kwak connected component contained 125M of edges, so its connectivity has almost doubled in 6 years. The average path length for our graph is 3.7 which is very close to the 4.1 found by Kwak et al. (Figure 1). Likewise, the diameter (longest shortest path between two nodes) for their graph is 18 which is very close to the one we obtained (15). We also observed that our

¹<http://www.epinions.com/>

²<https://dev.twitter.com/rest/public>

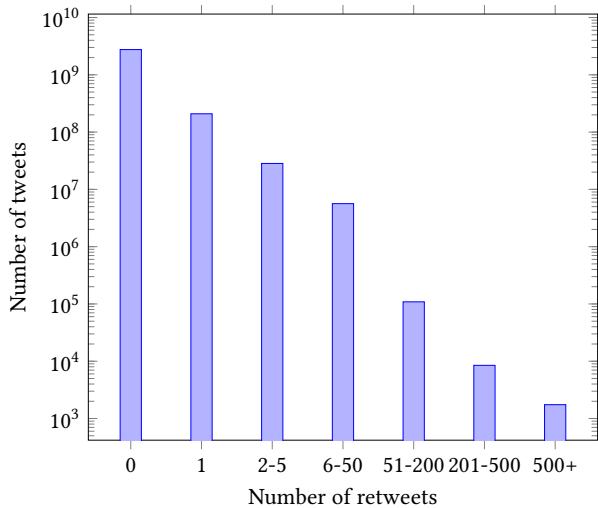


Figure 2: Distribution of the number of retweets per tweet

in/out-degree distributions (*i.e.* number of followers/followees per user) are close to the ones found in previous Twitter graph characterizations like [22, 35]. We conclude that we succeed in building a representative subgraph of Twitter along with the associated tweets published by the different accounts.

Based on our representative dataset we perform a set of experiments to analyze the retweet activity.

3.1 Retweet characterization

Our objective is to recommend relevant tweets to users. Retweet is the mean to express interest for a piece of information, which can indifferently be positively or negatively perceived. The study of tweet propagation through retweets is consequently of primary interest.

3.1.1 Retweet behavior. First, we analyzed the popularity of the retweet behavior. Figure 2 shows the number of retweets for a given message. A large majority of messages are never retweeted ($\approx 90\%$ of the tweets), or only a few times, with barely 2-3 retweets ($\approx 2\%$). Very popular messages are extremely rare: messages with more than 50 retweets represent less than 0.005%. These results are consistent with the study of Kwak [21] which underlines that a very large majority of the messages are never retweeted or retweeted only once. From a recommender system point of view, tweets which are retweeted more than 4-5 times must get a significant weight since they are popular. Observe that the network itself promotes the propagation of this type of messages (social network effect). However being able to recommend a recent tweet with a small number of retweets (less than 5) is a real challenge. This is one of the objective of our proposal.

When analyzing whether retweeting is a common behavior, we observe that only a small number of users produce many retweets (see Figure 3). The majority of users perform between 10 and 100 retweets. The average number of retweets per user is 156 while the median is 37.5 and we observe the traditional power law distribution where few people gather all the retweets. From the point of view of recommender systems, the main difficulty lies in users with very few retweets (or even none) who represent a large part of the users (a quarter of users have never retweeted). For these users, methods which rely on collaborative filtering would be unable to provide recommendations.

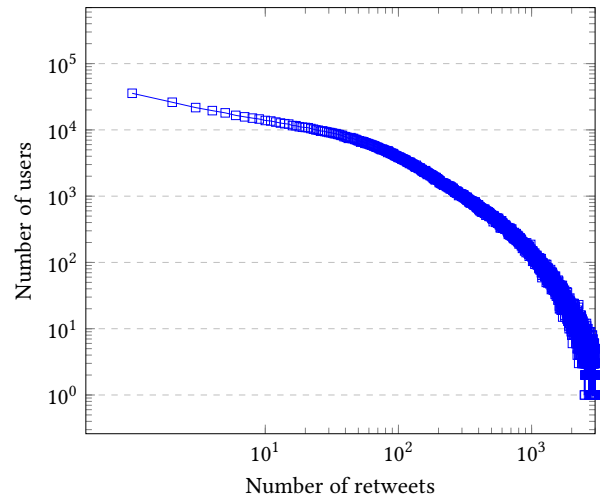


Figure 3: Number of retweets per user

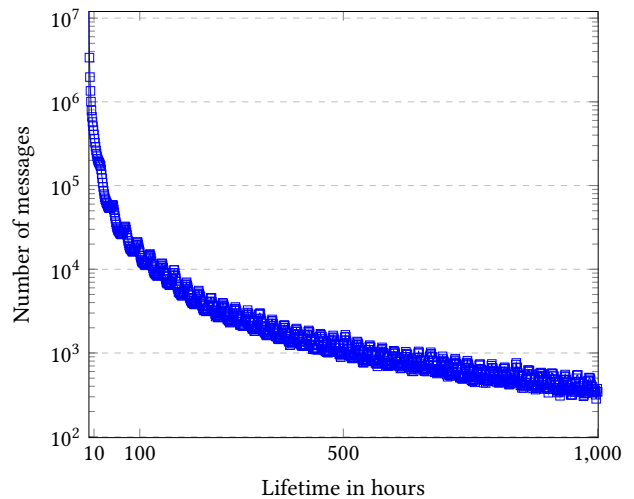


Figure 4: Lifetime of a tweet

3.1.2 Lifetime of a tweet. Recommender systems must avoid recommending a tweet which is “outdated”, *i.e.* which won’t be propagated anymore because it no longer interests anyone. We study here the lifetime of a tweet that we measured as the time span between the publication of the initial message and the last time it was retweeted. Only messages which were retweeted at least once are considered in this experiment. The results are shown in Figure 4. We observe that a large part of the messages “die” (so are no longer retweeted) before reaching one hour (40%). 90% of the messages die before 72h (three days). It’s very uncommon for a message to be retweeted beyond that point. Therefore our results differ from the ones reported by Kwak et al. [21] which notice that 10% of the retweet activity takes place one month after the publication of the message. Indeed, Twitter still was, in 2009, a recent system and less content was created, thus it was more easy to retweet messages older than a few weeks. In 2017, with more than 500 million new tweets published every day, freshness of the tweets has become a central criterion of propagation.

We conclude from this analysis that users mainly share fresh information and the tweets to consider in our recommender system are the messages that reach at least 2 retweets. Moreover, even for these messages, we do not need to compute their score after 72h because after this limit they become irrelevant for recommendation.

3.2 Homophily study

Collaborative filtering methods rely on a list of *similar* users to determine recommendations. We study in this section the evolution of similarity scores between users with respect to their distance in the Twitter network. We adopt a Jaccard similarity measure with a slight adjustment to take into account the popularity of the tweet as advised by Breese et al. in [4]: the less popular a retweeted post by two users is, the closer these users are. The rationale for relying on retweets to measure the similarity between users, as explained before, is that retweets are the only way a user expresses explicitly an interest about a piece of information published.

Definition 3.1 (User similarity measure). The similarity of two users is defined as:

$$sim(u, v) = \frac{\sum_{i \in L_u \cap L_v} \frac{1}{\log(1+m(i))}}{|L_u \cup L_v|} \quad (1)$$

where $sim(u, v)$ stands for the similarity score between a user u and a user v . L_u is the profile of user u estimated through the set of all the tweets he retweeted. $m(i)$ is the popularity of the tweet, measured here as the number of times it has been retweeted.

Based on this measure we study the homophily between users. Homophily corresponds in social media to the tendency for people to be connected with people sharing the same interest. This effect has been studied on demographic dimensions (age, gender, political orientation) for example by Colleoni [5] or Zamal [37]. They show how one could predict a user demographic based on his neighborhood. Topical homophily also has been studied by Lerman [15], that shows how people who are topically more similar are also more likely to be connected. Bhattacharya et al. [3] present similar results. With our analysis we study the homophily phenomenon in a different perspective.

Because of computation costs, we limited our study to 2,000 users randomly selected from our dataset, checking that they retweeted at least a defined number of posts. For each of these users, we computed the length of the shortest path connecting them to all other users. The results are shown in Table 2. We see that only 5.96% of user pairs display a similarity score over 0 and are directly connected through the network. However this small number of users pair presents the highest mean similarity score with 0.0056. When exploring at distance 2, to reach the followees of the followees, we note that the mean similarity score of 0.0021 is still higher than the average value (0.0019), and they represent almost 38% of the user pairs. Most of the users (51%) with a similarity score not null are at distance 3 from each other. However their similarity score is lower, 0.0017. Moreover, we observe that, according to the topology of the Twitter network (Figure 1), a distance of 3 corresponds almost to the entire network.

Table 3 completes this experiment by studying the link between position in the Top- N of most similar users and their distance to the user. For each of the 2,000 users from the previous experiment, we collect the 5 users with the highest similar scores. We then compute the average shortest distance between a user

Distance	Nb of users	Perc.	Average similarity
1	19,163	05.96%	0.0056
2	121,857	37.91%	0.0021
3	166,633	51.84%	0.0017
4	12,070	03.76%	0.0018
5	297	00.09%	0.0016
6	6	00.01%	0.0019
Impossible	1,396	00.43%	0.0023

Table 2: Evolution of the similarity score through distance in the network

Rank	Average Distance	Distances distribution (%)			
		1	2	3	4
1	1.65	53.30	28.20	16.65	1.45
2	1.78	43.70	34.50	20.50	1.05
3	1.88	37.99	36.04	24.37	1.35
4	1.97	33.18	36.99	27.68	1.70
5	1.99	32.01	37.93	28.20	1.56

Table 3: Link between distance in the network and position in the Top- N ranking

and each user from his Top-5. We observe that the user at first rank, *i.e.*, the most similar user, is directly connected to the user in 53% of the cases. The average score decreases when going down in the user rank. For instance the user ranked at the fifth position is directly connected to the user in only 32% of the cases. This study of the top-5 reveals that when considering users at a distance 1 added with users at distance 2 we capture 70-80% of the most similar users.

From this experiment we conclude that relying on the "strong" homophily, *i.e.* direct connection between users, is not enough because they represent a very small subset of the set of similar users. On the other hand we observe a "soft" homophily corresponding to users having a good similarity score but located at distance 2. These users which represent around 37% of users with a not null similarity scores, must consequently be considered by recommender systems to determine meaningful recommendations.

4 OUR MODEL

Based on our previous analysis we propose a propagation model relying on homophily to build post recommendation. Our approach relies on the construction of a similarity graph on top of the Twitter graph. The basic idea is to exploit the fact that highest similar users are at a distance lower or equal to 2. Thus we can use an exploration of the network to drastically reduce the amount of computed similarities.

4.1 Similarity graph

Our experiments about the homophily enlighten that the natural homophily in Twitter, which is translated by a "follow" link, is not sufficient to detect users of interest while considering all nodes at a distance up to 2 allows to capture most similar users. So, we decide to perform a 2-hop exploration from each node (user) in the Twitter graph. The set of reachable nodes for a node u is named the 2-hop neighborhood, denoted $\mathcal{N}_2(u)$. Then we compute for each node v in $\mathcal{N}_2(u)$ the similarity score of v with all the nodes from $\mathcal{N}_2(u)$. For each node w with a similarity greater

	SimGraph
Nb of nodes	1,149,374
Nb of edges	4,950,417
Mean Similarity Score	0.0078
Mean out-degree	5.9
Diameter	21
Mean smallest path	7.5

Table 4: SimGraph characteristics

than a chosen threshold τ , we create an oriented edge $e(u, w)$ in our similarity graph.

So formally the definition of the similarity graph is:

Definition 4.1 (Similarity graph). Consider the Twitter graph $G(V, E)$ where V denotes the set of vertices and E the set of edges, and a given similarity threshold τ . The similarity graph $SimGraph(V', E')$ is defined as:

$$\begin{cases} V' \subseteq V \\ e(u, v) \in E' \Rightarrow u \in V \wedge v \in N_2(u) \wedge sim(u, v) \geq \tau \end{cases}$$

We report in Table 4 the main characteristics of the similarity graph for our Twitter dataset. First, we observe that around half of the users from the original Twitter dataset are not present in the similarity graph. The reason is that many users either don't retweet any message or nobody has retweeted the same messages as this user. This issue is very close to cold-start problems and we won't address in this paper the issue of recommending tweets for users who are not present in the similarity graph.

However, we could consider an approach similar to the one used in *GraphJet* [32] using the neighborhood's computed recommendation of cold start nodes to partially solve this issue. Therefore it seems possible without too much effort to recommend items also for cold-start users.

Another characteristic of the similarity graph is a more uniform in-degree distribution than the original graph, with few nodes having a large number of incoming edges. This property is particularly interesting since in the opposite situation it would have led to some users gathering all the influence in the similarity network. Paths between two nodes in our similarity network are now much longer than the original Twitter network since the diameter of the similarity graph is now 21 and the average smallest path was doubled from 3.7 to 7.5. We observe that we are still in a small world network context according to the definitions expressed in the work of Schnetler [31]. Basically, this construction of a similarity graph, relying on 2-hop exploration in the Twitter graph could be seen as a dimension reduction process.

4.2 Propagation Model

The sparsity of user interactions on data is generally considered a major bottleneck for a successful execution of a collaborative filtering recommender system. As we've seen before, in microblogging context, this sparsity issue is important. In their work, Huand et al. [14] use transitivity in a collaborative filtering model to fight the sparsity problem. Considering a similar approach, we propose a propagation model on top of our similarity graph. Propagation is an efficient way to face the sparsity since it allows to transmit content from a user u to a user v while there exists no link (*i.e.*, edge in the underlying graph) between them. Intuitively we consider that if a user w has interests similar to u

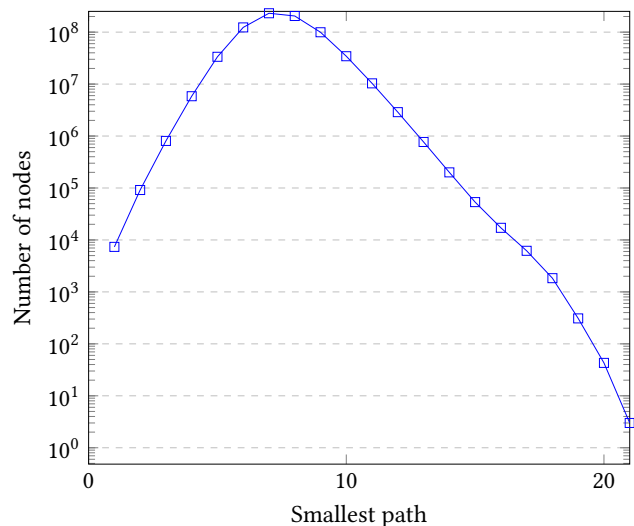


Figure 5: SimGraph smallest path

leading to several retweets, the content from u that we recommend to w should also be recommended to v if v is similar to w .

The only action we can collect to capture interests from a user for a tweet is the retweet action to share a piece of information. Consequently we assume here that liking a tweet is similar to sharing it (retweet). Therefore the words "like" and "retweet" are interchangeable. We estimate the sharing probability, *i.e.* the probability that a user u likes a tweet t as:

Definition 4.2. Sharing probability The probability that a user u likes (and shares) a tweet t is:

$$p(u, t) = \frac{\sum_{v \in F_u} p(u \leftarrow v, t)}{|F_u|}$$

where F_u denotes the set of influential (similar) users for u (*i.e.*, those connected by outgoing edges in the similarity graph) and $p(u \leftarrow v, t)$ is the probability that u likes t determined consistently by the behavior of his influential user v . This probability is estimated as:

$$p(u \leftarrow v, t) = p(v, t) \times sim(u, v)$$

Example 4.3. Consider the similarity graph of Figure 6 with 5 nodes (u, v, w, x, y). An edge $u \rightarrow v$ expresses the fact that v is an influential user of u . Edges values correspond to the similarity score $sim(u, v)$. Assume that user x liked/shared the tweet t_1 , so the probability that x like t_1 is $p(x, t_1) = 1$. Consequently the score of $p(w, t_1)$ is:

$$p(w, t_1) = \frac{\sum_{v \in F_w} p(w \leftarrow v, t)}{|F_w|} = \frac{0 \times sim(w, y) + 1 \times sim(w, x)}{2} = 0.25$$

However since $p(w, t_1)$ changes, all probabilities which depend on the value of $p(w, t_1)$ must be updated in turn.

5 PROPAGATION ALGORITHM

We introduce in this section our propagation algorithm which allows to recommend new content to users based on their similarity. We also present its convergence property and optimizations.

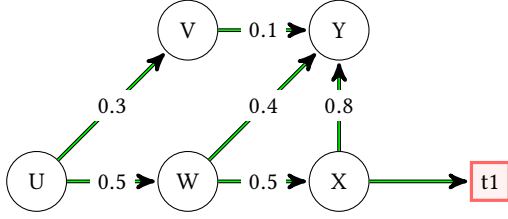


Figure 6: Example of similarity graph

5.1 Iterative algorithm

We perform our iterative propagation algorithm each time a tweet t is retweeted. We discuss different optimizations in Section ?? which exploit our observations regarding the popularity and the lifespan to avoid performing the propagation for each retweet.

So assume that we have a similarity graph $SimGraph = (V, E)$ with V the set of vertices and E the set of edges, and that a tweet t is retweeted by a user. Algorithm 1 presents our propagation algorithm. We denote by D the set of users that have already retweeted t . For each $v \in D$, the probability $p(v, t)$ is consequently 1 (line 3). For all other users $u \in V \setminus D$ we consider that the initial probability to retweet this tweet is $p(u, t) = 0$ (line 4). During an iteration, we compute for all the users u in $V \setminus D$ the probability $p(u, t)$ based on the probabilities from his followees, i.e. $p(v, t)$ for all $v \in F_u$ (line 10). So for iteration k we compute our probabilities based on the those which were computed at iteration $k - 1$. Observe that users $v \in D$ have a probability of 1 which is not re-computed during the iterative algorithm. The algorithm stops when no probabilities change during an iteration (line 7).

Algorithm 1: Propagation algorithm

input : Similarity graph $SimGraph = (V, E)$, a tweet t and a set of users D who retweet it

output: the set of vertices with their retweet probability

```

1 Initialization;
2 foreach u in V do
3   if u ∈ D then p(u, t) = 1;
4   else p(u, t) = 0;
5 end
6 convergence = false;
7 while convergence = false do
8   convergence = true;
9   foreach u in V \ D do
10    p'(u, t) = (∑_{v ∈ F_u} p(v, t) × sim(u, v)) / |F_u|
11    if p(u, t) ≠ p'(u, t) then convergence = false;
12    p(u, t) = p'(u, t);
13  end
14 end
15 Return ∀u ∈ V, p(u, t);

```

Example 5.1. Consider our previous example in Figure 6. After that user x liked/shared the tweet $t1$, we propagate this action on the similarity graph and we first update the score of $p(w, t)$ since x is an influential user of w . Since the value of $p(w, t)$ changed and is now 0.25 we must update in a second iteration the probability

of the users influenced by w , so here only u .

$$p(u, t1) = \frac{\sum_{v \in F_u} p(u \leftarrow v, t)}{|F_u|} = \frac{0 \times sim(u, v) + 0.25 \times 0.5}{2} = 0.0625$$

Since there is no users influenced by u the algorithm stops.

Therefore this algorithm produces for an incoming tweet a probability score for every users in the propagated sub-graph. Then, the recommendations of a user is based on the top- k tweets sorted on probability. We will study in Section 6.2 the impact of the size (k) on the quality of the recommendations compared to competitors.

5.2 Linear system formulation

Consider we have n users (u_1, u_2, \dots, u_n) in the similarity graph, our propagation model consists in resolving the following linear system with n equations:

$$\begin{cases} a_{11}p_{u_1} + a_{12}p_{u_2} + \dots + a_{1n}p_{u_n} = b_1 \\ a_{21}p_{u_1} + a_{22}p_{u_2} + \dots + a_{2n}p_{u_n} = b_2 \\ \dots = \dots \\ a_{n1}p_{u_1} + a_{n2}p_{u_2} + \dots + a_{nn}p_{u_n} = b_n \end{cases}$$

which can be written under a matrix product $Ap = b$ with:

- vector b which is the initial state vector, with $b_i = 1$ if u_i has retweeted (shared) the message and 0 otherwise;
- vector p is the solution vector, containing the probabilities after propagation of any user to like the tweet;
- the similarity matrix A which is defined as:
 $\forall i \leq n, \forall j \leq n,$

$$a_{i,j} = \begin{cases} 1 & \text{if } i = j \\ \frac{-sim(u_i, u_j)}{|F_{u_i}|} & \text{if } (u_i, u_j) \text{ is an edge in the } SimGraph \\ 0 & \text{otherwise} \end{cases}$$

5.3 Convergence property

Incremental resolution methods such as Jacobi, Gauss-Seidel or successive over-relaxation (SOR) can be used to solve such a linear system $Ap = b$. A necessary and sufficient condition to ensure convergence for this incremental resolution method is that the matrix A is diagonally dominant. This condition is fulfilled here because $\forall u, v \ sim(u, v) \leq 1$, so $\sum_{j \neq i} |a_{ij}| < \frac{1}{|F_u|} \times \sum_{j \neq i} 1 = 1$. Since $|a_{jj}| = 1$, we conclude that $|a_{jj}| \geq \sum_{j \neq i} |a_{ij}|$ for all i , so A is diagonally dominant.

Considering the Jacobi method of resolution, the convergence speed is bound by the matrix norm $\|A\|$. However, this value is dependent of the matrix's values, and therefore cannot be known theoretically. We conducted an experimental study on our dataset and show that the convergence of our model is bound to $\|A\| = 0.91$ - the worst case scenario. The unpredictable or bad convergence speed of our model led us to apply several optimizations to guarantee a fast convergence and therefore a fast computation.

5.4 Propagation algorithm optimizations

We propose and test several optimizations for the propagation algorithm which significantly improve its performance.

Propagation thresholds. A first optimization consists in setting a static threshold β to decide whether a score updated at an iteration k should be transmitted to the followees at the iteration

$k + 1$. So when $p(u, t)^n - p(u, t)^{n-1} < \beta$ the user u does not propagate its score to his followees for any following iteration.

We extend this traditional approach by proposing a dynamic threshold based on our analysis about the evolution of the tweet popularity and its lifespan. Indeed the probabilities computation for a popular message requires a long time in order to reach convergence. Moreover, this tweet is likely to be sent to a large part of our similarity network. Oppositely, a tweet which has not been retweeted many times because it has just appeared in the system leads to faster computations. Most recommender systems focus on already-popular tweets. So, by setting a dynamic threshold which favors tweets less popular because they have just appeared, we could recommend them earlier than other recommender systems, and at lower computing costs. Precisely we define our dynamic threshold $\gamma(t)$ for a tweet t as:

$$\gamma(t) = \frac{(m(t))^p}{k^p + (m(t))^p}$$

where $m(t)$ denotes the popularity of the message t which can be measured as the number of retweets, for instance. k and p are fixed parameters and are superior to 0, they are used to fit properly the distribution of popular items. Basically $\gamma(t)$ is bounded between $[0, 1]$ and close to 0 when few people have shared t , and close to 1 when the message is considered popular. However other dynamic thresholds could be considered.

Postponed computation. We propose another optimization based on the time frame update. It consists in starting the propagation process not at each time a new retweet happens on a message but after time interval δ depending on the account activity. For instance, assume that a message is very popular with dozens of retweets per minute. We could decide to wait 10 minutes before executing the propagation computation. On the opposite a very unpopular message can wait few hours before launching the propagation process.

6 EXPERIMENTS

We evaluate the *SimGraph* model along with our propagation algorithm. We first detail the experiment protocol used with our *Twitter* dataset to measure the quality of generated recommendations. Then we present our different results and compare them with other recommendation methods.

6.1 Experimental Setup

Our experiments are performed on a Linux machine with 512GB of RAM memory and 80 Intel(R) Xeon(R) CPU E7-4830 v2 @ 2.20GHz. We use Java Openjdk 1.8 for all our implementations. To compare our method with several baseline solutions, we implement a *Bayes* Inference Model used for recommendation [36], a standard collaborative filtering method (*CF*) [13] and *GraphJet* algorithm [32] which is currently used by *Twitter*. The rationale for the choice of these different competitors is that our method could be seen as a combination of probabilistic and collaborative filtering approaches. Regarding the Bayesian inference model we tweak it slightly to consider only the binary feedback of *Twitter* (liking or doing nothing) instead of ratings from 1 to 5. Moreover due to computational issues it is necessary to define a threshold in the Bayesian probabilities computation to stop the costly process. The implementation of the *GraphJet* method is based on *Twitter* source code³. All experiments are performed on the real *Twitter* dataset introduced in Section 3.

³<https://github.com/twitter/GraphJet>

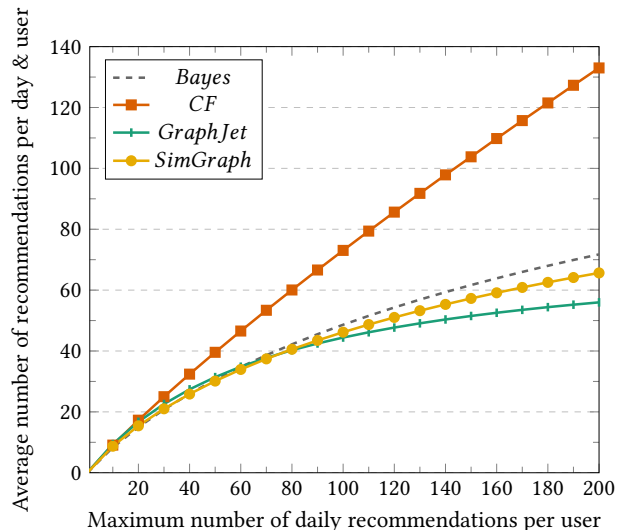


Figure 7: Recall Capacity for 1500 users

To measure the quality of the recommendations on our dataset, we compare the recommendations of the different methods with real observations. To achieve this, we consider retweets of messages which were retweeted at least twice. They constitute a set of 132, 389, 409 sharing actions for these messages that we ordered on time. We split the set in two: the first 90% of actions (the oldest retweets) compose the training set and the last 10% the test set. While the former set is used to train the four methods, the latter one allows to check the recommendations with real retweets. Note that the test set captures 66 days of retweets from the users in our dataset. Then we randomly select 500 users with less than 100 retweets (*low-active users*), 500 users with more than 100 retweets but less than 1,000 (*moderate-active users*) and finally 500 users with more than 1,000 retweets (*intensive users*). Combined, they constitute a set of 1,500 users used to compare the results of the different methods. We consider that a message is a *hit* if it is recommended to a user *before* he actually interact with the message (retweet/like). This prediction task can be seen as a quality measure.

6.2 Quality of the recommendations

Number of recommendations. Figure 7 displays the average number of recommendations proposed by the different recommender systems with respect to the maximum number of daily recommendations for each user (i.e. the size of the top- k recommendations). We observe that *Bayes*, *GraphJet* and *SimGraph* present similar behavior, capped between 50 and 70 recommendations per day and user. Only *CF*, with a linear growth, is capable of providing a high number of recommendations for any user with this maximum of 140 recommendations. There are several possible explanations for this. First, *CF* by relying on every similarity scores possible between pair of users is independent of the network. Therefore the scope of candidate items to be proposed is very large. Second, *Bayes* and *SimGraph* which rely on propagation methods could theoretically compute a prediction score for any message to almost any user. The fact that their curve is not linear seems to be a direct consequence of using thresholds during the propagation. We could argue that our threshold values are well chosen since 50 recommendations a day per user

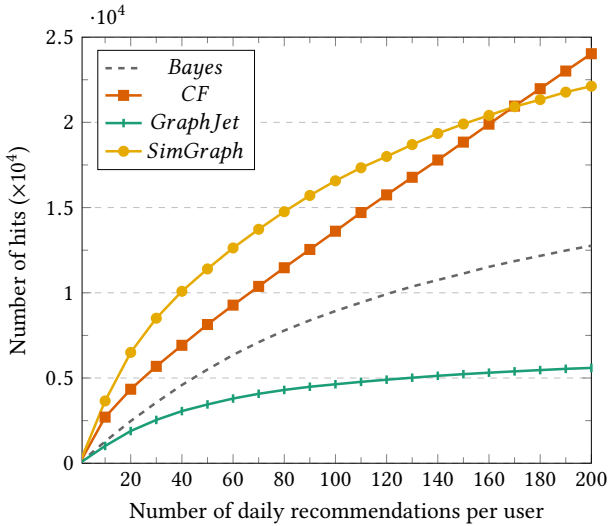


Figure 8: Number of hits for 1500 users

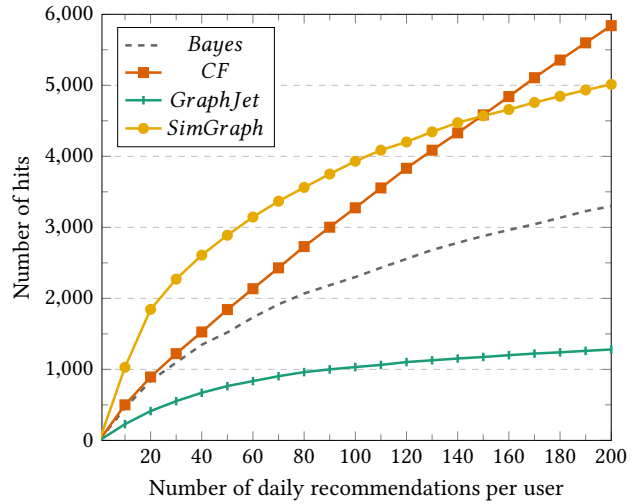


Figure 10: Number of hits for 500 *medium* users

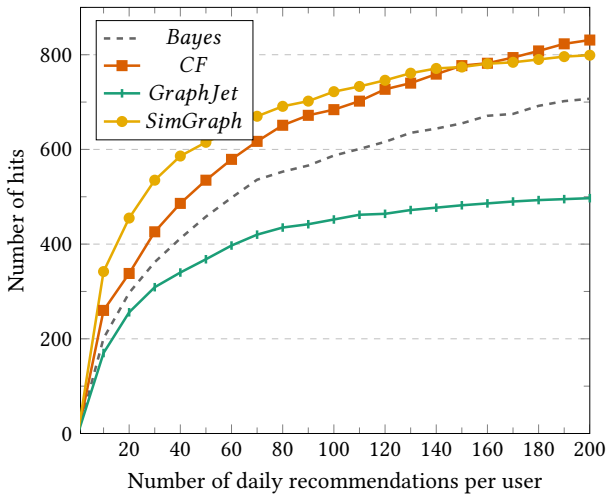


Figure 9: Number of hits for 500 *small* users

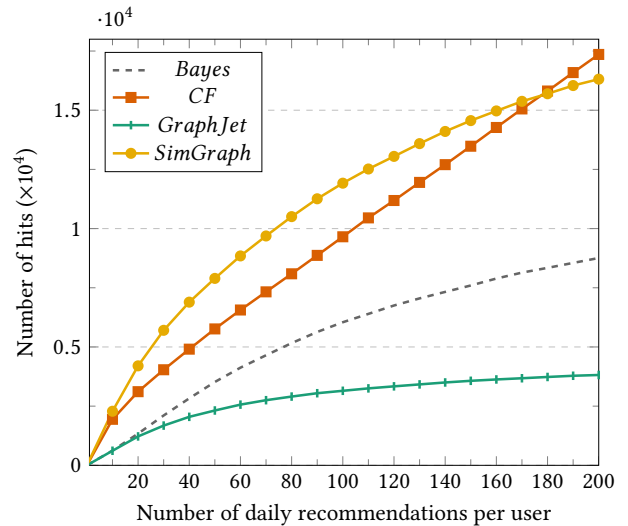


Figure 11: Number of hits for 500 *big* users

remains sufficient. Finally, *GraphJet* is limited by the low connectivity and the slow activity of small users. In fact, the total number of possibilities offered to small users is limited to their neighborhood.

Retweet prediction. A prediction corresponds to a hit in the test dataset when the recommendation of a message happens *before* it is effectively retweeted/shared in the dataset. Figure 8 plots the total number of hits for the combined set of 1,500 users with respect to the number of recommendations proposed to a user per day. This experiment is refined by different sets of users on Figure 9 (low-active users), Figure 10 (moderate-active users) and Figure 11 (big users). We observe that the results are globally stable among the different sets of considered users.

The main difference between types of users (small, medium, big) comes from the bounds of the number of hits. In fact, the total number of retweets/share is higher for big users and therefore the probability to have a hit. But interestingly, users' behavior is identical for medium and big users. According to small users, since the total number of retweets is low, the probability to find a

hit grows up quickly for all approaches but a threshold is reached faster.

We note that the *CF* model has a linear evolution in the combined set of 1500 users, so it is better suited for applications proposing a high number of recommendations. But as we saw in Figure 7 this model also proposes the largest number of recommendations which leads to a constant but low precision score of 0.0001% (*i.e.*, it proposes a lot of noise). For instance, when limiting to a top-30 for recommendations, *CF* gets 5,685 hits while *Bayes* and *GraphJet* get respectively 3,564 and 2,541 hits. *SimGraph* outperforms them with a total number of 8,509 hits at top-30. However, this number of hits is limited for small users with a stabilization of 700 hits for $k > 80$. It is due to the fact that those users do not retweet much and cannot statistically lead to much more hits.

SimGraph outperforms other approaches, *e.g.* *GraphJet* by a factor of 3.5, for any $k < 200$. For very large values of k , *CF* slightly outperform our method *Simgraph*. Similarly to *CF*, *SimGraph* does not rely directly on the underlying network and

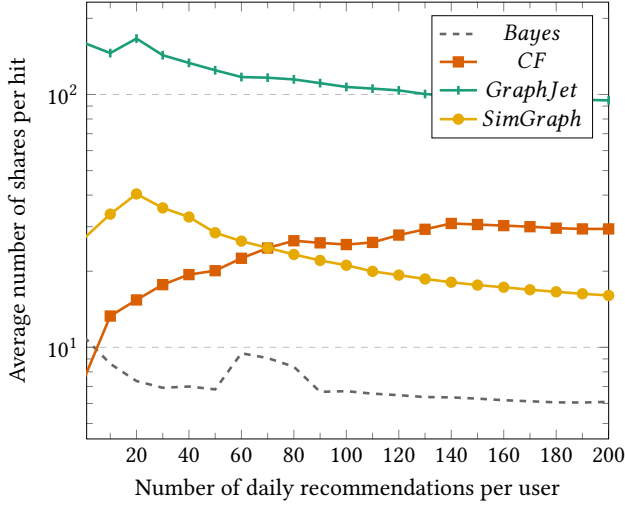


Figure 12: Popularity of hits

focuses on similarities which quickly provides good results based on users interests. But, thanks to the similarity graph and transitivity, our solution succeeds in providing new recommendations coming from other kinds of interests, different from the ones observed in the user profile, based on proximity where *CF* fails.

Popularity of the hits. We now focus on how popular the recommendations are in each solution which led to some hits. It helps to identify their scope of accuracy. Figure 12 shows that *GraphJet*'s random walks mainly generate hits on popular messages with an average number of retweets equals to 113 for each hit. Indeed, the most popular a message is, the more often the random walk will reach it. Therefore, *GraphJet* naturally is more inclined to recommend popular items.

On the contrary, the Bayesian model recommends less popular messages: for the hits it produced there were only 6 retweets on average. Thus, it produces more "local" recommendations, due to the probability computation on top of the underlying network. Observe however that the messages popularity decreases with the size of the top- k , in order to provide more recommendations *Bayes* will find items that are even less popular.

Collaborative filtering approaches, *CF* and *SimGraph*, present a more balanced result with respectively 35 and 23 retweets for a recommended tweet. These systems recommend both popular and more dedicated messages. Observe that at first *CF* will propose less popular items due to relying on strong similarities far in the network while *SimGraph* will propose more popular items. The curve will then intersect around top-70.

Hits comparison. Figure 13 displays the amount of hits in common between *SimGraph* and other methods. It shows the ratio σ of common hits, i.e. :

$$\sigma(\text{competitor}) = \frac{\text{hits}(\text{SimGraph}) \cap \text{hits}(\text{competitor})}{\text{hits}(\text{competitor})} \quad (2)$$

We notice first that the ratio of common hits does not evolve more than 10% for each method. Except for *CF* that proposes less popular retweet at the beginning (see Figure 8) and more popular ones then, which levels up the number of common hits.

On the contrary, *GraphJet* focuses mainly on popularity at the beginning and shares less similarity between users, but then

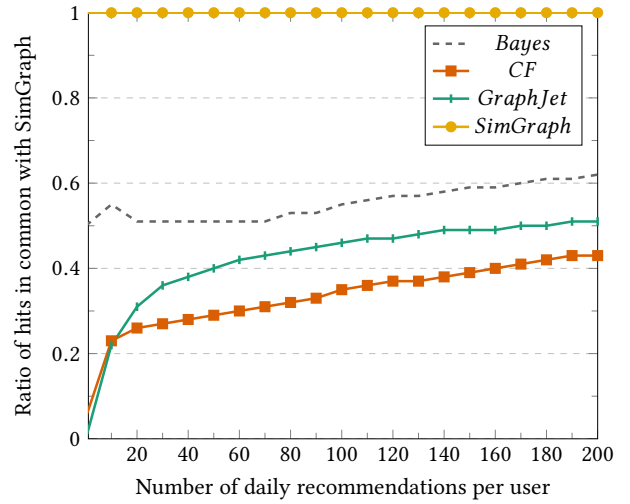


Figure 13: Parts of hits that are included in SimGraph

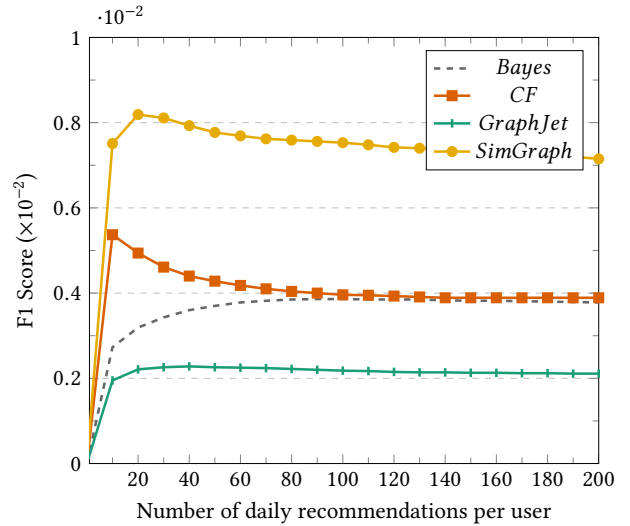


Figure 14: F1 Scores over number of recommendations

after 40 recommendations this intersection reaches a bound due to a more diversified set of recommendations in *SimGraph*.

The Bayesian method shares more similarities between hits with more than 50% which means that our method predicts retweets of tweets with various popularity: popular tweets like *GraphJet*, but also less popular tweets like *CF* and not-popular like *Bayes* based on the network.

F1 Measure. Figure 14 plots the F1 score for each method, according to the hits obtained over the number of daily recommendations per user. Except *Bayes*, methods peak around the same size of $k = 15$, which seems to be a good number of recommendation per day. *GraphJet* by doing less hits for same amount of recommendation leads to a lower score when combining precision and recall. *CF* proposes too many recommendations even with better precision obtains a similar F1 score as *Bayes* that proposed less recommendations but with more hits. Finally, *SimGraph* gives a really good compromise, with both popular and

similar in the neighborhood, of recommendations of tweets. Overall on this task, *SimGraph* performs 4 times better than *GraphJet* and 2 times better than *Bayes* and *CF*.

6.3 Performances

Processing time. We compare now processing time for the different methods. Results are presented in Table 5. We observe that *CF* has the highest initialization time with almost 9s per user, since it requires to compute the similarities between all pairs of users, so theoretically $|V|^2 \approx 1.3 \times 10^{12}$ computations. Oppositely *Bayes* has the lowest initialization time with 10ms per user to initialize all probabilities. For *SimGraph*, the initialization consists in exploring for each user his neighborhood up to distance 2 (BFS exploration) and to compute similarities for each user found during the exploration to build the similarity graph. For each user, using this method requires 311ms to determine its most similar users. *GraphJet* only relies on the Twitter and retweets graph and does not need any initialization.

According to the message processing time, all the methods were implemented in a multi-thread way on a 70 cores parallelization process. We can observe that *Bayes* takes around 1 second to compute the recommendation scores for a given message due to graph exploration. Oppositely *CF* is able to compute very quickly the different recommendations (0.5ms per message) based on the pre-computed similarities. *SimGraph* requires 38ms to compute the recommendation score of a message to users based on our iterative propagation algorithm. *GraphJet* is user-centric and computes top- k recommendations for a given user, and not a set of users to recommend a given message. Computing k recommendations for a given user requires 14ms.

To compare performances of the different algorithms we measure the total time for processing both the initialization and the recommendations, for each user for *GraphJet* or for each of the 13.2 million incoming messages during the test dataset for other methods. For *GraphJet* we perform the computation periodically, here we chose every 5 hours. Note that the test period extends over 66 days. It results that *Bayes* is the most expensive method with 51.22h of total computation. *CF* is also expensive with 41.01h of computation mainly for the pre-processing. *GraphJet* provides much faster recommendations (4.2h). *SimGraph* is a good trade-off with an initializing time by node of 311ms and an average time per message of 38ms leading to a total computation time of 3.41h. We see that both the *Bayes* and *CF* methods suffer of very long computation time, making them hard to use in a real life scenario.

Notification time. This experiment illustrates the benefit of the different methods regarding the notification time. In other words, we want to study how much time in advance a recommended tweet will be presented to the user before the actual interaction. Remember that according to our study in Section 3, a tweet has a short lifespan. We compute for each hit the difference between the time when the recommendation was performed and the time when the user perform the retweet in the test dataset. In Figure 15, we see that *GraphJet* is very stable and permits to predict a hit 80,000s (around 22 hours) in advance on average. *GraphJet*'s tendency to recommend popular items makes it more efficient on this task, having more opportunities to predict such messages before. Interestingly the *CF* curve is correlated with the average popularity of the item predicted in Figure 12. Recommending dedicated items long time in advance is a difficult task therefore, and naturally the average advance time increases for *CF* and

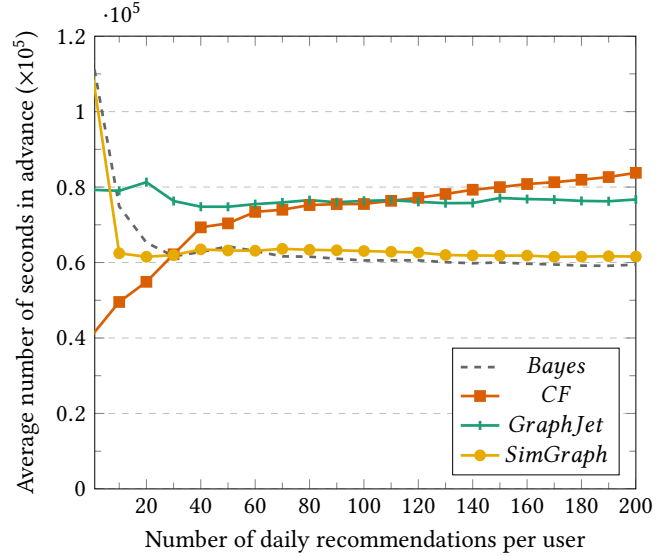


Figure 15: Average advance time before real retweets (in s)

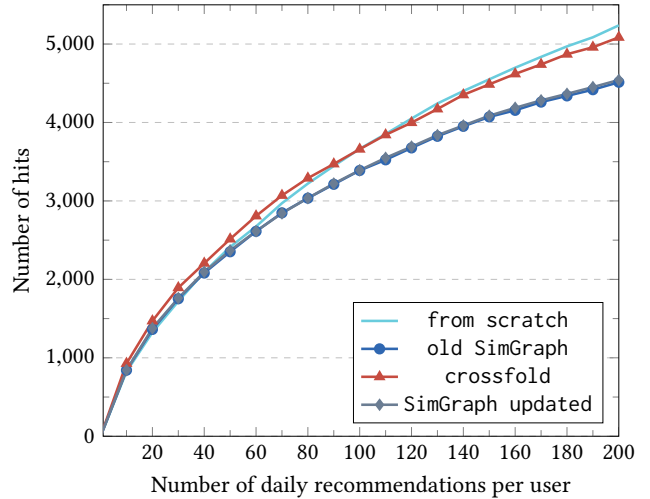


Figure 16: Number of hits with several updating strategies

can predict recommendations before *GraphJet* for very large k values. Indeed *CF* benefits from not having to wait for a message to propagate through the network to compute a recommendation score, therefore can predict popular items long time in advance. Following the same conclusion that predicting dedicated messages long time in advance is a difficult task, it fully explain why *Bayes* and *SimGraph* can only predict iterations 17h in advance on average since they have to wait for more signals.

Graph update. Microblogging platforms such as Twitter are permanently in motion. Each retweet/like performed impacts some changes of similarity scores and consequently impacting recommendations computation and quality. Keeping structures up to date to fit the shift of users' interest is a very difficult task. Therefore an efficient updating strategy is crucial to make our recommendation model robust to network and retweet evolutions. *GraphJet* is a very suitable solution in real life because, by avoiding any initialization step, it continuously stays updated no matter the amount of new information published. However, as

	init. (per user)	init total time	time (per message)	total time (70 cores //)	total time
		1,149,374 users		13,238,941 Tweets (Trial period)	init + recos
Bayes	10ms	0.04h	975ms	51.22h	51.26h
CF	8,583ms	39.40h	0.5ms	0.02h	41.01h
SimGraph	311ms	1.41h	38ms	2.00h	3.41h
	init. (per user)	init total time	time (per user)	total time (70 cores //)	total time
		1,149,374 users		1,149,374 users * 66 days (Trial period)	init + recos
GraphJet	0ms	0h	14ms	4.2h	4.2h

Table 5: Initialization and recommendation time (in ms)

we saw in Figure 8 the gain in hits prediction with our method *SimGraph* is substantial enough to inquire how we can keep this increase of accuracy while dealing with incremental updates. We study the impact of computing our recommendations based on an outdated similarity graph and the outcome of using different updating strategies. For this experiment we assume that *SimGraph* was built at the beginning of the test dataset (after 90% of the retweets). We plot in Figure 16 the number of hits obtained for the last 5% of the retweets (half of the 10% trial). We compare four approaches:

- from scratch, where the graph is totally rebuilt after 95% of the retweets, from the original graph,
- old *simgraph*, where we keep the exact same *SimGraph* that the one computed at 90%,
- *crossfold*, where we apply our similarity graph construction on the previous old *simgraph* instead of the twitter graph,
- *SimGraph* update, where we update similarity scores on the similarity graph built at 90%

As expected re-building from scratch the similarity graph at 95% allows to get the best predictions. This strategy is also the most expensive one with similar computation cost as the ones expressed in Table 5. Surprisingly old *SimGraph* and *SimGraph* update have almost the exact same results, indicating that the topology of the computed network has a more important impact than the weight computed on the edges. The *crossfold* strategy is very promising because it fits almost perfectly with the from scratch strategy while cutting drastically computation cost. Indeed, the *crossfold* strategy perform a BFS at distance 2 on the already computed similarity graph to search for new influential users that weren't considered during the first *SimGraph* computation. This method both increases the density of the graph while updating the weight edges. These results enlighten the possibility to follow the evolution of users by incrementally computing a *SimGraph* on top of the previous iteration and avoid computing things from scratch.

7 CONCLUSION

We propose in this paper *SimGraph*, a scalable recommendation model based on a similarity graph which exploits homophily for propagation of probabilities. We study the homophily impact between users on the network. This indicate that homophily is a good property to find quickly users of high similarities and therefore drastically reduce the computation cost of such operation. We find that applying transitivity on those similarities reduces the sparsity efficiently with a fast convergent model thanks to optimization technics. We propose a propagation model in order to compute on-demand relevant recommendations for an incoming post. Our experiments enlighten that our method outperforms

other approaches for recommendation computations, but also provides more hits than the competitors for a lower number of recommendations blending popular and more confidential items. We also demonstrate how *SimGraph* can be updated at low cost, showing its usability in real world scenarios.

As future works, we intend to enhance our similarity graph by analyzing content of the tweets with entity recognition. In fact, our similarity is based on common retweets between users and can be improved by creating "topic tweets" by merging similar tweets. This will make users likely to be similar in the similarity graph and therefore enhance results for small users. We also plan to break "information bubbles", since recommended information is generally originated from the same sub-part of the graph. We are currently working on the identification of bubbles in our twitter graph based on both the network topology and tweet topics. Then we will propose a complementary score for recommendations by escaping from information locality from a bubble to another.

REFERENCES

- [1] Gediminas Adomavicius and Alexander Tuzhilin. 2005. Toward the Next Generation of Recommender Systems: A Survey of the State-of-the-art and Possible Extensions. *Trans. on Knowledge and Data Engineering (TKDE)* 17, 6 (2005), 734–749.
- [2] Hyung Jun Ahn. 2008. A New Similarity Measure for Collaborative Filtering to Alleviate the New User Cold-starting Problem. *Inf. Sciences* 178, 1 (2008), 37–51.
- [3] Parantapa Bhattacharya, Muhammad Bilal Zafar, Niloy Ganguly, Saptarshi Ghosh, and Krishna P. Gummadi. 2014. Inferring User Interests in the Twitter Social Network. In *Proc. Intl. Conf. on Recommender Systems (RECSYS)*. 357–360.
- [4] John S. Breese, David Heckerman, and Carl Kadie. 1998. Empirical Analysis of Predictive Algorithms for Collaborative Filtering. In *Proc. Intl. Conf. on Uncertainty in Artificial Intelligence (UAI)*. 43–52.
- [5] Elanor Colleoni, Alessandro Rozza, and Adam Arvidsson. 2014. Echo Chamber or Public Sphere? Predicting Political Orientation and Measuring Political Homophily in Twitter Using Big Data. *Jour. of Communication* 64, 2 (2014), 317–332.
- [6] Camelia Constantin, Ryadh Dahimene, Quentin Grossetti, and Cédric Du Mouza. 2016. Finding Users of Interest in Micro-blogging Systems. In *Proc. Intl. Conf. on Extending Database Technology (EDBT)*. 5–16.
- [7] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep Neural Networks for YouTube Recommendations. In *Proc. Intl. Conf. on Recommender Systems (RECSYS)*. 191–198.
- [8] Peter Forbes and Mu Zhu. 2011. Content-boosted Matrix Factorization for Recommender Systems: Experiments with Recipe Recommendation. In *Proc. Intl. Conf. on Recommender Systems (RECSYS)*. 261–264.
- [9] Blaž Fortuna, Carolina Fortuna, and Dunja Mladenić. 2010. Real-time News Recommender System. In *Proc. Eur. Conf. on Machine Learning and Knowledge Discovery in Databases (ECML-PKDD)*. 583–586.
- [10] Frédéric Godin, Viktor Slavkovikj, Wesley De Neve, Benjamin Schrauwen, and Rik Van de Walle. 2013. Using Topic Models for Twitter Hashtag Recommendation. In *Proc. Intl. World Wide Web Conference (WWW)*. 593–596.
- [11] David Goldberg, David Nichols, Brian M. Oki, and Douglas Terry. 1992. Using Collaborative Filtering to Weave an Information Tapestry. *Commun. ACM* 35, 12 (1992), 61–70.
- [12] Pankaj Gupta, Ashish Goel, Jimmy Lin, Aneesh Sharma, Dong Wang, and Reza Zadeh. 2013. WTF: The Who to Follow Service at Twitter. In *Proc. Intl. World Wide Web Conference (WWW)*. 505–514.

- [13] Jonathan L Herlocker, Joseph A Konstan, Al Borchers, and John Riedl. 1999. An algorithmic framework for performing collaborative filtering. In *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, 230–237.
- [14] Zan Huang, Hsinchun Chen, and Daniel Zeng. 2004. Applying Associative Retrieval Techniques to Alleviate the Sparsity Problem in Collaborative Filtering. *ACM Trans. Inf. Syst.* 22, 1 (Jan. 2004), 116–142.
- [15] Jeon hyung Kang and Kristina Lerman. 2012. Using Lists to Measure Homophily on Twitter. In *AAAI work. on Intelligent Techniques for Web Personalization and Recommendation*.
- [16] Mohsen Jamali and Martin Ester. 2010. A matrix factorization technique with trust propagation for recommendation in social networks. In *Proceedings of the fourth ACM conference on Recommender systems*. ACM, 135–142.
- [17] Meng Jiang, Peng Cui, Rui Liu, Qiang Yang, Fei Wang, Wenwu Zhu, and Shiqiang Yang. 2012. Social Contextual Recommendation. In *Proc. Intl. Conf. on Information and Knowledge Management (CIKM)*. 45–54.
- [18] Danae Pla Karidi, Yannis Stavrakas, and Yannis Vassiliou. 2017. Tweet and Follower Personalized Recommendations Based on Knowledge Graphs. *Jour. of Ambient Intelligence and Humanized Computing* (2017), 1–15.
- [19] Yehuda Koren. 2008. Factorization Meets the Neighborhood: a Multifaceted Collaborative Filtering Model. In *Proc. Intl. Conf. on Knowledge Discovery and Data Mining (KDD)*. 426–434.
- [20] Yehuda Koren, Robert Bell, and Chris Volinsky. 2009. Matrix Factorization Techniques for Recommender Systems. *Computer* 42, 8, 30–37.
- [21] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a Social Network or a News Media?. In *Proc. Intl. World Wide Web Conference (WWW)*. 591–600.
- [22] Kristina Lerman, Rumi Ghosh, and Tawan Surachawala. 2012. Social Contagion: An Empirical Study of Information Spread on Digg and Twitter Follower Graphs. *CoRR* abs/1202.3162 (2012).
- [23] Pasquale Lops, Marco De Gemmis, and Giovanni Semeraro. 2011. Content-based Recommender Systems: State of the Art and Trends. In *Recommender Systems Handbook*. Springer, 73–105.
- [24] Hao Ma, Haixuan Yang, Michael R Lyu, and Irwin King. 2008. Sorec: Social Recommendation Using Probabilistic Matrix Factorization. In *Proc. Intl. Conf. on Information and Knowledge Management (CIKM)*. 931–940.
- [25] Paolo Massa and Paolo Avesani. 2007. Trust-aware Recommender Systems. In *Proc. Intl. Conf. on Recommender Systems (RECSYS)*. 17–24.
- [26] Arthur Mensch, Julien Mairal, Bertrand Thirion, and Gaël Varoquaux. 2016. Dictionary Learning for Massive Matrix Factorization. In *Proc. Intl. Conf. on Machine Learning (ICML) (ICML'16)*. 1737–1746.
- [27] Bradley N. Miller, Istvan Albert, Shyong K. Lam, Joseph A. Konstan, and John Riedl. 2003. MovieLens Unplugged: Experiences with an Occasionally Connected Recommender System. In *Proc. Intl. Conf. on Intelligent User Interfaces (IUI)*. 263–266.
- [28] N.Koumchatzky and A.Andryeyev. 2017. Using Deep Learning at Scale in Twitter’s Timelines. (2017). <https://blog.twitter.com/2017/using-deep-learning-at-scale-in-twitter-s-timelines>
- [29] Aäron van den Oord, Sander Dieleman, and Benjamin Schrauwen. 2013. Deep Content-based Music Recommendation. In *Proc. Intl. Conf. on Neural Information Processing Systems (NIPS)*. 2643–2651.
- [30] Alan Said and Alejandro Bellogin. 2014. Comparative Recommender System Evaluation: Benchmarking Recommendation Frameworks. In *Proc. Intl. Conf. on Recommender Systems (RECSYS)*. 129–136.
- [31] Sebastian Schnetzler. 2009. A Structured Overview of 50 Years of Small-World Research. *Social Networks* 31, 3 (2009), 165 – 178.
- [32] Aneesh Sharma, Jerry Jiang, Praveen Bommanavar, Brian Larson, and Jimmy Lin. 2016. GraphJet: Real-time Content Recommendations at Twitter. *Proc. VLDB Endow.* 9, 13 (2016), 1281–1292.
- [33] Jiliang Tang, Xia Hu, and Huan Liu. 2013. Social recommendation: a review. *Social Network Analysis and Mining* 3, 4 (2013), 1113–1133.
- [34] Ibrahim Uysal and W Bruce Croft. 2011. User Oriented Tweet Ranking: a Filtering Approach to Microblogs. In *Proc. Intl. Conf. on Information and Knowledge Management (CIKM)*. 2261–2264.
- [35] Jianshu Weng, Ee-Peng Lim, Jing Jiang, and Qi He. 2010. TwitterRank: Finding Topic-sensitive Influential Twitterers. In *Proc. Intl. Conf. on Web Search and Data Mining (WSDM)*. 261–270.
- [36] Xiwang Yang, Yang Guo, and Yong Liu. 2013. Bayesian-Inference-Based Recommendation in Online Social Networks. *IEEE Trans. Parallel Distrib. Syst.* 24, 4 (2013), 642–651.
- [37] Faiyaz Al Zamal, Wendy Liu, and Derek Ruths. 2012. Homophily and Latent Attribute Inference: Inferring Latent Attributes of Twitter Users from Neighbors.. In *ICWSM*.

Online Set Selection with Fairness and Diversity Constraints

Julia Stoyanovich*
Drexel University
Philadelphia, PA
stoyanovich@drexel.edu

Ke Yang
Drexel University
Philadelphia, PA
ky323@drexel.edu

HV Jagadish†
University of Michigan
Ann Arbor, MI
jag@umich.edu

ABSTRACT

Selection algorithms usually score individual items in isolation, and then select the top scoring items. However, often there is an additional diversity objective. Since diversity is a group property, it does not easily jibe with individual item scoring. In this paper, we study set selection queries subject to diversity and group fairness constraints. We develop algorithms for several problem settings with streaming data, where an online decision must be made on each item as it is presented. We show through experiments with real and synthetic data that fairness and diversity can be achieved, usually with modest costs in terms of quality.

Our experimental evaluation leads to several important insights in online set selection. We demonstrate that theoretical guarantees on solution quality are conservative in real datasets, and that tuning the length of the score estimation phase leads to an interesting accuracy-efficiency trade-off. Further, we show that if a difference in scores is expected between groups, then these groups must be treated separately during processing. Otherwise, a solution may be derived that meets diversity constraints, but that selects lower-scoring members of disadvantaged groups.

1 INTRODUCTION

Diversity is desired in many contexts, ranging from results of a Web search to admissions at a university. As algorithms are increasingly used to make decisions, there is growing interest in algorithms that can produce diverse results. Indeed, fairness and diversity are central to responsible data science practice [7, 17].

Diversity is a set concept: it makes no sense to talk about an individual item as being diverse. Fairness is less clearly a set concept; nevertheless, fairness is often stated with respect to some comparison standard, usually a group [5, 12, 19]. For example, in the context of racial discrimination, we frequently refer to under-represented minorities, which is a set construct, with fairness requiring proportional representation.

Most algorithmic decision-making is based on the individual: typically, a score is assigned to an individual item based on its attributes. However, since fairness and diversity are set concepts, they can only be guaranteed as part of a set selection procedure.

In this paper, we show how we can guarantee fairness and diversity in set selection. We begin by developing a simple general problem statement in Section 2, to maximize *utility* subject to a set of *diversity constraints*. We show that our problem formulation covers a wide range of fairness and diversity requirements. We then solve this problem in two settings. In Section 3, we present a baseline algorithm that make the assumption that all items are available before any selections have to be made. Then, in Section 4

we develop algorithms that decide whether to accept, reject or defer an item in an online manner, as the items are presented. We refer to this variant as the Diverse K -choice Secretary Problem.

Algorithms of Section 4 constitute the main technical contribution of this paper. These algorithms build upon a rich body of work on the Secretary Problem [8, 11, 14] — selecting the maximum element in a randomly-ordered sequence of N elements, and on its K -choice variant — selecting K elements out of N [4].

In Section 5, we show experimentally that the online algorithms of Section 4 produce solutions that both meet the diversity requirements and are very close to the baseline algorithm of Section 3 in terms of utility. Further, we demonstrate that theoretical guarantees on solution quality of online algorithms are conservative on real datasets. These algorithms start by observing the scores of the items in the stream without accepting any items, to develop a quality estimate; this is known as the warm-up period. We show that an interesting quality-efficiency trade-off can be achieved by tuning the length of the warm-up period. Finally, we show that if a difference in scores is expected between groups, then these groups must be treated separately during processing. Otherwise, a solution may be derived that meets diversity constraints, but that results in selecting lower-scoring members of historically disadvantaged groups.

We discuss related work in Section 6 and conclude in Section 7.

2 PROBLEM DEFINITION

The basic problem setting is that we have a set of items, each with associated attributes. From this set, we wish to select K items to maximize a utility score (to be defined below) subject to diversity constraints (also to be defined below). The items in the set may be presented to us together or one at a time.

We obtain the utility score for a set of K selected items as the sum of scores of each individual selected item. The *score* of an item may be pre-computed and stored as a physical attribute, or it may be computed on the fly, and possibly even be obtained as the result of an expensive scoring algorithm. In all cases, all we require is that we eventually have a single scalar score value for each item. The score is sometimes called the *utility score* or *utility value* in the literature.

The basic top- k problem is to choose K items with the highest score. That is, for any item j in the top- k , and any other item q not in the top- k , we have $s_j \geq s_q$, where s_q is the score of item q . This is equivalent to saying we choose $K \geq 0$ items such that $\forall k \in [0, K][\text{argmin}_{j \in [0, K]}(s_j)]$ is maximized. This is further equivalent to saying $\sum_{j \in [0, K]} s_j$ is maximized. We will use this last definition, since with added diversity constraints these three definitions are no longer equivalent, and the first two may not be appropriate.

Having described the utility maximization problem above, let us now turn to fairness and diversity constraints. Among the attributes associated with items, we assume that one discrete-valued attribute is of particular concern. We call this the *sensitive*

*This work was supported in part by NSF Grants No. 1741047 and 1464327.

†This work was supported in part by NSF Grants No. 1250880 and 1741022.

attribute. Our notions of fairness and diversity are defined with respect to the value of this sensitive attribute.

In practice, there may be multiple sensitive attributes, rather than just one. In this case, we could consider each independently, by making minor appropriate modifications to all statements below. If combinations of multiple attributes are of concern, or if dependencies between the sensitive attributes need to be captured explicitly, we could represent such combinations as a single (Cartesian product) attribute of concern. For example, if both race $\in \{W, B, H\}$ and gender $\in \{M, F\}$ are sensitive attributes, we could combine these into a single attribute of cardinality 6.

If a sensitive attribute is not discrete-valued, or takes on too many discrete values, then we can bucketize the attribute value into a finite number of discrete buckets. Attributes such as age and salary are often treated this way in practice for many applications. In fact, sensitive attributes may also have associated privacy concerns, and so may need to be converted to noisy histograms, e.g., to enforce differential privacy.

We further assume that the dataset is partitioned on the value of the sensitive attribute. That is, each item is associated with exactly one value of the sensitive attribute. For example, a person of mixed race should not be listed as having both White and Black as values for the race attribute: rather this value should be set to an appropriate single value, such as "White-Black-Mixed".

Let there be d distinct values of the sensitive attribute. Our requirement is to choose k_i elements for each distinct value $i \in [1..d]$, with each $k_i \in [0, K]$, and $\sum_i k_i = K$. Of course, this begs the question of what the k_i values should be. We next consider several notions of fairness and diversity and show how to capture these within this framework.

Fairness by proportional representation (of values of the sensitive attribute). Suppose that the number of items N is known, as is the number of items n_i in each sensitive category $i \in [1..d]$. Then, proportional representation requires that the desired size K of the selected set be prorated among the d categories. That is $k_i = K * n_i / N$. We call the right hand side of this equation $proportion_i$, for convenience.

A difficulty we run into is that k_i must be an integer: an item in some category is either selected or it is not. Thus, fractional values do not make sense, yet $proportion_i$ is not always an integer. We can round $proportion_i$ to the nearest integer to determine each k_i , hoping to return K items in total. But we may end up with rounding errors resulting in violation of $\sum_i k_i = K$. To avoid this, it is reasonable to provide some flexibility in choosing the value of each k_i , using the formula $\lfloor proportion_i \rfloor \leq k_i \leq \lceil proportion_i \rceil$, where $\lfloor . \rfloor$ is the floor function and $\lceil . \rceil$ is the ceiling function.

Even weaker constraints are often acceptable in practice. For example, in a class of 821 students, and with a binary assignment of the gender attribute, we may desire to see 410 students of one gender and 411 of the other. However, it is unlikely that an institution would be accused of discrimination if they admitted 407 women and 414 men. Generally, it is acceptable to set thresholds on the relative representation of different categories. This idea is a generalization of the 80% rule of disparate impact [10].

Another potentially appropriate fairness metric is the normalized difference: the mean difference normalized by the rate of positive outcomes, which in our case corresponds to being selected among the top- k . Another is the elift ratio: the ratio of positive outcomes for the historically disadvantaged demographic group over the general group. A ratio of 1 indicates no discrimination, while a ratio below 0.8 has been construed as discrimination by

US courts. These and other proportional representation metrics can be found in a recent survey by Zliobaite [19].

Coverage-based diversity. A popular measure of diversity is coverage [7]: is there representation for every category in the selected set? Whether this is possible depends on how K , the number of items selected in total, compares to d , the number of categories of items. If $d \geq K$, then each $k_i \leq 1$. We cannot get full coverage, but by not choosing 2 from any category, we make sure to include a representative from as many categories as possible. If $d \leq K$, then each $k_i \geq 1$. Since K is large enough in this case, we can have multiple items from each category as long as we make sure that we have at least one from each category.

To avoid "tokenism" — selecting a single representative of each category, we may want to specify coverage diversity in terms of a larger minimum number per category. For example, we may require that there be at least 5 members of each race in the selected set. Such a choice would typically be made only if $5d \leq K$, and our requirement becomes that each $k_i \geq 5$.

Summarizing the scenarios considered above, we can state the specific diversity or proportionality constraint of interest as $floor_i \leq k_i \leq ceil_i$, where $floor_i$ and $ceil_i$ are integers that are determined, for each i , based on the particular constraint of interest. This formulation allows us to treat combinations of sensitive attributes (represented by a single Cartesian product attribute) in a way that captures attribute dependencies. For example, we can derive the constraint for the number of female candidates of a minority race to be higher or a lower than what would result from $proportion_{female} \times proportion_{minority}$.

The general statement of our problem is as follows:

Diverse Set Selection Problem Statement: Given N items, each with an associated utility score and an identified sensitive attribute, for each value i of the sensitive attribute, choose k_i items such that the summation utility of the selected set is maximized, subject to $floor_i \leq k_i \leq ceil_i$ and subject to $\sum_i k_i = K$. The $floor_i$ and $ceil_i$ values depend on the specific constraint to be applied. These values are computed prior to the optimization problem, and are assumed to be given.

All N items may be given together; we call this the static case and study it in Section 3. Alternatively, the items may arrive one at a time; we call this the online case and study it in Section 4.

The standard cost-metric in the top- k problem is the number of items examined: ideally, this should be much less than N . We carry over this metric to our problem domain as well. This metric, which we call *walking distance* (it is sometimes called *depth* in the top- k literature), is a simple surrogate for the incurred CPU cost, and has the advantage of being independent of the implementation and of the execution environment. We will discuss in Section 4 that walking distance relates to solution utility in the online case, and so is more informative than wall-clock time.

Another standard top- k cost metric is *buffer size*: the in-memory storage cost for running the algorithm. We do not present experimental results on buffer size, but note that all algorithms proposed here use buffers of constant size, under the assumption that K and $\sum_i ceil_i$ are constants.

Finally, as we shall see when we get to the online algorithms, we cannot always get the best answer if we are required to decide for each item on the spot. An *accuracy* metric we develop will reflect how close the online solution comes to the true optimum. We note that this optimum is the best we can do subject to the



Figure 1: An illustration of the static scenario. $N = 12$ items, labeled a through l, belong to one of two classes, blue and red. The goal is to select $K = 3$ items subject to $1 \leq k_{blue} \leq 2$ and $1 \leq k_{red} \leq 2$. Items arrive in score-sorted order, with scores ranging from 9 down to 1.

diversity or fairness constraints: a higher score may be possible without these constraints. We describe all metrics in Section 5.3.

3 THE STATIC PROBLEM

In this section, we solve the problem for the case when we have access to all items. We call this the *static* case. In the next section, we will turn to the online (streaming) case.

In the traditional set up for the top- k problem with multi-attribute criteria, the problem setting assumes that we have items sorted by attributes of interest, with our ranking criterion being some monotone aggregation function of these attribute values (e.g., weighted sum). We proceed down the sorted list(s), stopping when we can predict that an unseen item cannot possibly be included in the selected set [9].

In our problem setting, item scores are precomputed, and so we consume a single list of items, sorted by decreasing score. But we have a more complex selection criterion: Diversity constrains each k_i , the number of items with a value i for the sensitive attribute, to $floor_i \leq k_i \leq ceil_i$.

Recall that d is the number of distinct values of the sensitive attribute. Let us define $required = \sum_{i=1}^d floor_i$. For our set of floor and ceiling constraints to be feasible, we must have $required \leq K$. The difference $K - required = slack$, represents the total slack that we have to choose items after all floor constraints are satisfied. To return a set of items with the highest utility (total score), we are best off filling the slack with items of highest utility, unconstrained by sensitive attribute value, as long as the number of items per category does not exceed the respective ceiling constraint. We use this observation in Algorithm 1. We illustrate the algorithm with an example.

Example 3.1. Consider the score-sorted list of items in Figure 1. $N = 12$ items are partitioned into $d = 2$ categories (blue and red), with 6 items per category. The goal is to select $K = 3$ items, with between 1 and 2 items per category. That is, $floor_{red} = floor_{blue} = 1$ and $ceil_{red} = ceil_{blue} = 2$.

We process items in order, left-to-right. At step 1, blue item a is accepted to meet the $floor_{blue}$ constraint. Among the remaining 2 items that will be accepted, one must be red, to meet the $floor_{red}$ constraint, and the other can be of either color. At step 2, blue item b is encountered and accepted. At this point, only one item remains to be accepted, and it must be red. At step 3, blue item c is skipped. Finally, at step 4, red item d is accepted, meeting the $floor_{red}$ constraint, and selecting the required $K = 3$ items. The algorithm terminates after consuming 4 items.

Let us now consider the pseudocode of Algorithm 1. As illustrated in Example 3.1, the algorithm accepts an item if the floor constraint of its category has not been met (line 7), or if the ceiling constraint of its category has not been met and some

Algorithm 1 Diverse top- k selection from a sorted list.

Require: List of items I sorted by score,
number of items to select K , number of categories d ,
constraints $floor_i \leq k_i \leq ceil_i$ for each $i \in [1 \dots d]$.
{Initialize the output list L .}

- 1: $L = \emptyset$
{Initialize the counts of per-category selected items C .}
- 2: $C = [k_1 = 0, \dots, k_d = 0]$
{Compute the slack value s .}
- 3: $slack = K - \sum_{i=1}^d floor_i$
- 4: **while** $|L| < K$ **do**
- 5: $x = getNextItem(I)$
- 6: $i = category(x)$
- 7: **if** $k_i < floor_i$ **then**
- 8: $L \leftarrow x$
- 9: $k_i = k_i + 1$
- 10: **else if** $(k_i < ceil_i) \wedge (slack > 0)$ **then**
- 11: $L \leftarrow x$
- 12: $k_i = k_i + 1$
- 13: $slack = slack - 1$
- 14: **end if**
- 15: **end while**
- 16: **return** L

slack remains (line 10). Algorithm 1 terminates once K items are selected.

In general, we only have inequality constraints on each of the k_i values. However, note that in the special case that for each i , $floor_i = ceil_i$, we have the value of each k_i determined exactly. In this case, once the floor constraints are met for each category, we will have selected K items in total, since $K = \sum_i k_i = \sum_i floor_i$.

Algorithm 1 never examines any item with score lower than the smallest score included in the selected set. In this sense, the number of items examined is optimal — there is no way to examine fewer items if we proceed strictly in score order. This optimality result holds even though the worst case number of items examined is still N .

Besides the computational cost, we have one additional important notion of goodness to consider: that of the utility score. It is straightforward to establish the following Theorem.

THEOREM 3.2. *Algorithm 1 produces a solution that has the highest possible utility score, subject to the given constraints.*

Even though Algorithm 1 is optimal, subject to the diversity constraints, in general it will return K items with the combined utility score that is lower than would be possible in the absence of these constraints. This cost of diversity was illustrated in Example 3.1: we skipped item c although accepting it would maximize utility, but would result in selecting all items from the same category, blue. We will quantify the cost of diversity experimentally in Section 5 (Figure 7).

4 THE ONLINE PROBLEM

In practice, even though a set has to be selected, not all items in the set may be available for evaluation at once. Rather, they may appear one at a time, with a decision to be made on the specific item instantaneously. For example, we may wish to hire a diverse set of employees. However, each hiring decision may have to be made individually on each job applicant when the job application arrives. The order of arrival of applications is not, in general, determined by the quality of the applicants. More generally, we

have to classify each *individual* item, as presented, into one of two buckets: “selected” or “not selected,” subject to the utility and diversity criteria in our problem statement, for the selected *set*. Such situations motivate us to consider an online scenario, which is sometimes referred to as streaming.

Returning to the hiring example, we note that, while the quality of an applicant may be unknown ahead of the job interview, it is reasonable to assume that the number of applicants, both over-all and in each demographic category (e.g., by race, gender or some other sensitive attribute) can be known ahead of time, because these properties are declared by the applicants. The classic Secretary Problem and its variants, described next, and our proposed solution presented in the remainder of this section, rely on this information.

4.1 Background and Problem Statement

The problem of designing an online algorithm to optimize the probability of selecting the maximum element in a randomly-ordered sequence has been studied extensively [8, 11, 14], and is traditionally known as the Secretary Problem. In this problem, the goal is to hire one secretary from a pool of N candidates, where N is known, and candidates arrive in random order. When a candidate is interviewed, the decision must be made to hire or reject the candidate, and this decision is irreversible. It was shown by Lindley [14] and by Dynkin [8] that the optimal hiring strategy is to interview $m = \lfloor \frac{N}{e} \rfloor$ candidates without making any offers (this is called the *warm-up period*), and make an offer to the first candidate who is better than the best of the first m candidates (or accept the last candidate if no better candidate is seen). This strategy yields the best candidate with probability $\frac{1}{e}$, and is said to have *competitive ratio* e . Further, this is the best such strategy for the Secretary Problem, *i.e.*, with the highest competitive ratio [11].

A generalization of the Secretary Problem called the K -choice Secretary Problem is stated as follows: design an online algorithm for picking K out of N non-negative numbers presented in random order, to maximize their expected sum. While a straightforward extension of the Secretary Problem is natural here (with the same length of warm-up, $\lfloor \frac{N}{e} \rfloor$, remembering the scores of the K highest-scoring candidates), the exact optimal competitive ratio for this problem is not known for $K > 1$. This quantity is known to lie between $1 + c\sqrt{k}$ and $1 + C\sqrt{k}$ for some pair of constants $c < C$ [3].

Another interesting variant is the Poset Secretary Problem: If the elements of the permutation (candidates) are only partially ordered, how to maximize the probability of returning a maximal element in the poset? The incomparable elements present the main challenge: many simple modifications of the total order algorithm to handle incomparable elements were shown to have vanishing success probabilities [13].

In this section, we state, and then present a solution to, the online variant of the Diverse Set Selection Problem of Section 2:

Diverse K -choice Secretary Problem Statement: Design an online algorithm for picking K out of N items, each with an associated non-negative utility score and an identified sensitive attribute, presented in random order. Select items to maximize their expected sum, subject to diversity constraints of the form $floor_i \leq k_i \leq ceil_i$ for each value i of the sensitive attribute, and subject to $\sum_i k_i = K$.



Figure 2: An illustration of the online scenario. $N = 12$ items belong to one of two classes: blue and red, with $n_{blue} = n_{red} = 6$. The goal is to select $K = 3$ items subject to $1 \leq k_{blue} \leq 2$ and $1 \leq k_{red} \leq 2$.

4.2 Online Algorithm

We now present Algorithm 2 that solves the Diverse K -choice Secretary Problem. The basic idea of this algorithm is to solve d K -choice Secretary Problems [4] *in parallel*, one for each category, to satisfy the per-category *floor* constraints. But that in itself is not enough: we also have to run a category-insensitive K -choice Secretary algorithm to select the remaining items, subject to *ceiling* constraints.

Algorithm 2 relies on the estimates of the number of items per category in the stream (n_i represents the estimate of the number of items from category i), and guarantees that diversity constraints are met if these estimates are accurate. We now illustrate Algorithm 2 with an example.

Example 4.1. Consider the stream of items in Figure 2. $N = 12$ items are partitioned into $d = 2$ categories, with 6 items per category: $n_{red} = n_{blue} = 6$. Like in Example 3.1, the goal is to select $K = 3$ items subject to $1 \leq k_{blue} \leq 2$ and $1 \leq k_{red} \leq 2$. In contrast to Example 3.1, items arrive in random order.

To start, we compute the lengths of the per-category warm-up periods: $r_{blue} = \lfloor \frac{n_{blue}}{e} \rfloor = 2$ and $r_{red} = \lfloor \frac{n_{red}}{e} \rfloor = 2$. Therefore, we will consider, and discard, 2 items in each category before accepting any items in that category. As we consider the warm-up items, we record $floor_{blue} = 1$ highest blue item score, and $floor_{red} = 1$ highest red item score in the respective per-category threshold heaps T_{red} and T_{blue} .

Similarly, we compute the length of the category-independent warm-up period $r = \lfloor \frac{N}{e} \rfloor = 4$. The number of items we will accept irrespective of their category membership corresponds to the difference between K and the sum of the floor constraints, and is 1 in our example. (We called this quantity *slack* in Algorithm 1.) Therefore, we will record the score of the highest-scoring item (of any category) among the first $r = 4$ items in the threshold heap T , setting $T = \{8\}$ (the score of item d).

The warm-up period for the blue category will terminate at step 3, after items a and c are considered, with $T_{blue} = \{6\}$. At step 4, a blue item d is encountered, with score 8, higher than $getMinElement(T_{blue}) = 6$, and this item is accepted.

The warm-up period for the red category will terminate at step 5, after items b and e are considered, with $T_{red} = \{4\}$ (score of b). We will reject the next red item, g, because its score is lower than $getMinElement(T_{red})$, and will accept the following red item i at step 9 to satisfy $floor_{red}$.

We are also looking to accept an item with a score higher than $getMinElement(T) = 8$ from any category, as long as its ceiling constraint is not exceeded. However, because i (score 9) was used to satisfy $floor_{red}$, which takes precedence, no such item is encountered. To return K items, we must accept the last item in the stream, l with score 5.

We terminate with the output $\{d, i, l\}$, with utility $8+9+5 = 22$. This is only slightly lower than the best possible utility of 24.

Algorithm 2 Diverse K -choice Secretary Algorithm

Require: Stream of items I , total number of items to select K , input size N , number of categories d , constraints $floor_i \leq k_i \leq ceil_i$ and number of items per category n_i for $i \in [1 \dots d]$.
{Initialize the output list L .}

- 1: $L = \emptyset$
{Initialize the array of counts of per-category selected items C .}
- 2: $C = [k_1 = 0, \dots, k_d = 0]$
{Initialize counts of per-category seen items M .}
- 3: $M = [m_1 = 0, \dots, m_d = 0]$
{Compute the length of per-category warm-up.}
- 4: $R = [r_1 = \lfloor \frac{n_1}{e} \rfloor, \dots, r_d = \lfloor \frac{n_d}{e} \rfloor]$
{Initialize d MinHeaps, one per category, $T_1 \dots T_d$.}
- 5: **for** $i=1 \dots d$ **do**
- 6: $T_i = \text{MinHeap}(floor_i)$
- 7: **end for**
- 8: $slack = K - \sum_{i=1}^d floor_i$
{Compute the length of category-independent warm-up.}
- 9: $r = \lfloor \frac{N}{e} \rfloor$
{Initialize a category-independent heap T .}
- 10: $T = \text{MinHeap}(slack)$
- 11: **while** $|L| < K$ **do**
- 12: $x = \text{getNextItem}(I)$
- 13: $i = \text{category}(x)$
- 14: **if** $\sum_i m_i < r$ **then**
- 15: $T \xrightarrow{\text{offer}} x$
- 16: **end if**
- 17: **if** $m_i < r_i$ **then**
- 18: $T_i \xrightarrow{\text{offer}} x$
- 19: **else if** $((k_i < floor_i) \wedge (\text{score}(x) > \text{getMinElement}(T_i))) \vee$
 $(n_i - m_i == floor_i - k_i)$ **then**
- 20: $\text{deleteMinElement}(T_i)$
- 21: $L \leftarrow x$
- 22: $k_i = k_i + 1$
- 23: **else if** $(\sum_i m_i \geq r) \wedge (\text{score}(x) > \text{getMinElement}(T) \wedge$
 $(k_i < ceil_i) \wedge (slack > 0))$ **then**
- 24: $\text{deleteMinElement}(T)$
- 25: $L \leftarrow x$
- 26: $k_i = k_i + 1$
- 27: $slack = slack - 1$
- 28: **else if** $(k_i < ceil_i) \wedge (\text{numFeasibleItems}() == K - |L|)$
 then
- 29: $L \leftarrow x$
- 30: $k_i = k_i + 1$
- 31: $slack = slack - 1$
- 32: **end if**
- 33: $m_i = m_i + 1$
- 34: **end while**
- 35: **return** L

In Example 4.1, we happen to satisfy both floor constraints (at steps 4 and 9) before accepting a category-independent item at the end of the stream. Note, however, that this may not be the case in general. For example, we could have accepted a blue item with a score higher than 8, had one been encountered at steps 5, 6, 7, or 8 – any time after $floor_{blue}$ is met.

Processing of item d in Example 4.1 illustrates that different streams (per-category and category-independent) are consumed

in parallel: d is part of the category independent warm-up (where it is discarded but its score is recorded in T), and it is also part of the post-warm-up stream for the blue category, where it is accepted, since its score exceeds $\text{getMinElement}(T_{blue})$.

Let us now consider the pseudocode for Algorithm 2. The algorithm uses a MinHeap data structure to keep track of the top- K elements seen thus far. We need one for each category (denoted T_i and initialized on line 6 with capacity $floor_i$), and an additional one for the extra elements after the floor constraints have been met (denoted T and initialized on line 10 with capacity $slack$). Each per-category heap T_i stores the best $floor_i$ scores seen among the first r_i items of category i . If $floor_i > r_i$, then T_i will store the first $floor_i$ elements observed, together with $floor_i - r_i$ elements of value -1 . Heap T is initialized similarly, storing the best $slack$ scores seen among the first $r = \lfloor \frac{N}{e} \rfloor$ items, irrespective of category.

During the warm-up period, an item x is not accepted (not added to L) irrespective of its score, but rather is offered to the relevant per-category heap (on line 18) or to the category-independent heap (on line 15). Note that the same item x may be offered to its per-category heap and to the category-independent heap during warm-up.

An item of category i is added to the output after the warm-up period if $floor_i$ is not yet satisfied and either (a) the item has a sufficiently high score or (b) we are at the end of the stream for category i (line 19). The latter condition is evaluated by comparing the number of items remaining in the stream ($n_i - m_i$) to the number of items still required for i ($floor_i - k_i$).

An item is also added to the output if its score is sufficiently high according to the category-independent estimate and there is sufficient slack to meet all outstanding floor constraints (line 24). Note that Algorithm 2 uses the slack mechanism in a similar way as Algorithm 1.

Finally, an item is added to the output if it is *feasible*: accepting it would not violate the ceiling constraint for its category, and if exactly $K - |L|$ feasible items remain in the input. We compute the number of feasible items (line 28) as a sum of $n_j - m_j$ (the number of items that remain on the stream in category j) over all feasible categories (those in which $ceil_j - k_j > 0$).

This final set of conditions (line 28) is required to ensure that exactly K items are returned by Algorithm 2. Asserting that exactly $K - |L|$ feasible items remain in I relies on the estimates of the number of items in each category.

Optimality. Algorithm 2 specifies per-category lengths of the warm-up period on line 4. What is the competitive ratio of this algorithm? To reason about this, let us first consider the K -choice Secretary Problem, a generalization of the Secretary Problem where $K \geq 1$ rather than 1 item is to be chosen in an online manner. Recall from Section 4.1 that, when $K = 1$, the optimal competitive ratio is e , and it is achieved with a warm-up period of length $\lfloor \frac{N}{e} \rfloor$ [8, 14]. For $K > 1$, it is known that competitive ratio is no worse than e under the same warm-up period length [4], but the optimal competitive ratio is not known [3].

Our problem setting, and its solution presented in Algorithm 2, differ from the generalized K -choice Secretary Problem in that we are receiving items from multiple distinct categories. Algorithm 2 treats items that belong to different categories as different sub-streams of a common stream, and is guaranteed to have a competitive ratio no less than e for selecting $floor_i$ items in each category, by an immediate application of the result of Babaioff et al. [4]. The remaining $slack$ items are selected from the common

stream (subject to floor and ceiling constraints), and will have a competitive ratio no less than e (subject to the same constraints).

We will empirically compare the quality of the result returned by Algorithm 2 to that of the static algorithms of Section 3 in Section 5.4. We will also consider the impact of warm-up period length on accuracy in that section.

Impact of per-category warm-up on utility. An important point to note is that, by estimating scores on a per-category basis rather than for the entire set of items at once, Algorithm 2 accommodates the case when score is not independent of category membership. Consider an example in which there are two categories A and B , and where, for all pairs of items $a \in A, b \in B, \text{score}(a) < \text{score}(b)$. Suppose further that a and b occur in the input in approximately equal proportion. Then, if a common heap T of size K is maintained for both categories during warm-up (with $K < \lfloor \frac{N}{e} \rfloor$), T will contain scores of some K items from B , and so it will be the case that, at any point in time, $\forall a \in A, \text{getMinElement}(T) > \text{score}(a)$. As a result, an online algorithm will accept a subset of B with a high combined score, and it will accept floor_A items from A that appear at the end of the stream. This represents the worst case for category A in terms of utility. We validate this claim experimentally in Section 5.4.

4.3 Online Algorithm with a Deferred List

In a true on-line setting, a decision must be made whether to accept or to reject an item once it is seen. In practice, it may be acceptable to keep a waiting list of modest size. For example, college admissions work this way.

We now introduce Algorithm 3, an optimized version of Algorithm 2 that will often return a set of K items of higher utility, subject to diversity constraints. This is accomplished by introducing per-category deferred lists D_i of bounded size. We now give an intuition behind this algorithm using an example.

Example 4.2. Consider again the stream of items in Figure 2. $N = 12$ items are partitioned into $d = 2$ categories, with 6 items per category: $n_{red} = n_{blue} = 6$, and arrive in random order. The goal is to select $K = 3$ items subject to $1 \leq k_{blue} \leq 2$ and $1 \leq k_{red} \leq 2$. We now highlight the differences in the processing of this input when a deferred list is allowed, as compared to that of Example 4.1 (Algorithm 2).

We conduct a warm-up period of length 2 for each category, storing $\text{floor}_{blue} = \text{floor}_{red} = 1$ highest score in each $T_{blue} = \{6\}$ and $T_{red} = \{4\}$. In contrast to Example 4.1, we do not conduct a category-independent warm-up period (there is no T).

Also in contrast to Example 4.1, items encountered during the warm-up period are not immediately discarded, but rather are placed into per-category deferred lists D_{blue} and D_{red} , which maintain up to $\text{ceil}_{blue} = 2$ and $\text{ceil}_{red} = 2$ items, respectively. Even beyond the warm-up period, we consider the scores of all encountered items, and always keep 2 best items of the appropriate category seen so far in each D_{blue} and D_{red} . Note that D_{blue} and D_{red} are MinHeaps, which we denote by sorting the elements in the order of increasing score.

In our example, $D_{blue} = \{c, a\}$ at step 3 (end of warm-up for blue), $D_{blue} = \{a, d\}$ at step 4 (d has a higher score than c and replaces c), and $D_{red} = \{e, b\}$ at step 5 (end of warm-up for red). We continue processing until a sufficient number of items post the warm-up period is seen (1 post-warm up item in each category in our example) and once there are at least K items in the union of deferred lists. We continuously update the

Algorithm 3 Diverse K -choice Secretary Algorithm with a deferred list

Require: Stream of items I , total number of items to select K , input size N , number of categories d , constraints $\text{floor}_i \leq k_i \leq \text{ceil}_i$ and number of items per category n_i for $i \in [1 \dots d]$.

{Initialize the output list L .}

- 1: $L = \emptyset$
- {Initialize i deferred lists D_i of capacity ceil_i for each category.}
- 2: **for** $i=1 \dots d$ **do**
- 3: $D_i = \text{MinHeap}(\text{ceil}_i)$
- 4: **end for**
- {Initialize the list of counts of per-category selected items C .}
- 5: $C = [k_1 = 0, \dots, k_d = 0]$
- {Initialize the list of counts of per-category seen items M .}
- 6: $M = [m_1 = 0, \dots, m_d = 0]$
- {Compute the length of per-category warm-up.}
- 7: $R = [r_1 = \lfloor \frac{n_1}{e} \rfloor, \dots, r_d = \lfloor \frac{n_d}{e} \rfloor]$
- {Initialize d MinHeaps, one per category, $T_1 \dots T_d$.}
- 8: **for** $i=1 \dots d$ **do**
- 9: $T_i = \text{MinHeap}(\text{floor}_i)$
- 10: **end for**
- {Initialize the number of unsatisfied categories u .}
- 11: $u = d - \sum_{i=1}^d \mathbb{1}[\text{floor}_i == 0]$
- {Initialize the total number of deferred items w (for “waiting”).}
- 12: $w = 0$
- 13: **while** $(u > 0) \vee (w < K)$ **do**
- 14: $x = \text{getNextItem}(I)$
- 15: $i = \text{category}(x)$
- 16: **if** $m_i < r_i$ **then**
- 17: $T_i \xleftarrow{\text{offer}} x$
- 18: **else if** $(k_i < \text{ceil}_i) \wedge (\text{score}(x) > \text{getMinElement}(T_i))$ **then**
- 19: $k_i = k_i + 1$
- 20: $\text{deleteMinElement}(T_i)$
- 21: **if** $(\text{floor}_i > 0) \wedge (k_i == \text{floor}_i)$ **then**
- 22: $u = u - 1$
- 23: **end if**
- 24: **end if**
- 25: $D_i \xleftarrow{\text{offer}} x$
- 26: $m_i = m_i + 1$
- 27: $w = \sum_i |D_i|$
- 28: **end while**
- 29: $W = \text{MaxHeap}(w)$
- 30: $W \leftarrow \bigcup_i D_i$
- 31: Invoke Algorithm 1 on W (sorted by score), compute L .
- 32: **return** L

per-category deferred lists as we process, evicting lower-scoring items and keeping 2 highest-scoring items in each category.

In our example, the algorithm terminates after step 9 (item i), with $D_{blue} = \{a, d\}$ and $D_{red} = \{b, i\}$. The union of these lists is then passed to Algorithm 1, which returns $\{a, d, i\}$, with combined utility 23. Per Example 4.1, this utility is 1 point higher than of executing Algorithm 2 on this input.

We will illustrate experimentally in Section 5.4 that Algorithm 3 usually returns a set of K items of higher utility than Algorithm 2. Note, however, that Algorithm 3 may sometimes

return items of lower combined utility, because it may terminate sooner than Algorithm 2.

We now describe Algorithm 3 in detail. To start, set u (unsatisfied) to the number of categories with $\text{floor}_i > 0$. Add the first ceil_i items to D_i irrespective of their score, then maintain no more than ceil_i items in D_i , replacing the lowest-scoring item $y \in D_i$ with item x if $\text{score}(y) < \text{score}(x)$.

A category becomes satisfied once floor_i items in D_i have a sufficiently high score (post warm-up). If a sufficient number of high-scoring items cannot be found, we satisfy floor_i by adding the required number of items of category i from the end of its stream.

The algorithm stops consuming its input once all unsatisfied categories become satisfied (i.e., once $u == 0$) and the total size of all D_i is at least K . Note that, even after a category i is satisfied, we can still add item x to D_i (while waiting for the remaining categories to be satisfied), if x happens to have a higher score than the lowest-scoring item currently in D_i .

Having filled the deferred lists, the algorithm will first add floor_i items from each D_i to the output list L , and will then fill the remaining slack positions with the highest-scoring items from the remaining deferred lists, irrespective of their scores. Algorithm 1 can be invoked for this purpose, with $I = \bigcup_i D_i$.

Note that when Algorithm 1 is invoked on line 31, it is invoked on input W , a sorted list whose size is bounded by $\sum_i \text{ceil}_i$. We assume that $\text{ceil}_i \ll |I|$. In fact, setting any ceil_i higher than K is not meaningful. Further, since K is commonly treated as a constant, then $\sum_i \text{ceil}_i$ can also be treated as such.

5 EXPERIMENTAL EVALUATION

In this paper, we have introduced diversity and fairness constraints into set selection queries under several different settings. Most importantly, we have introduced two streaming algorithms. For all our algorithms we are interested in evaluating the cost of introducing a diversity or fairness constraint in terms of the lower utility achieved. For streaming algorithms, we are further interested in how well we manage to satisfy a group constraint and make a group selection, while being forced to make decisions regarding individual items as they are presented.

5.1 Experimental Datasets

Our experimental evaluation is conducted on both real and synthetic datasets. The real data gives us a sense for what would happen in a real scenario, while the synthetic data let us vary parameters to dive deeper into understanding "what if" questions.

Forbes Richest. We selected two Forbes Richest People lists from 2016: US Richest with 400 individuals (<https://www.forbes.com/forbes-400/list/>) and World's Richest with 526 individuals (<https://www.forbes.com/billionaires/list/>). Both lists are naturally ranked by net worth. We used gender as the sensitive attribute in the US list, with a break-down of 27 female vs. 373 male individuals ($d = 2$ categories). We used country as the sensitive attribute in the World list, creating separate categories for US (197 individuals), Germany (44), China (43), Russia (25), and assigning the remaining 217 individuals to the category "other", resulting in $d = 5$ categories.

NASA Astronauts. This dataset is available at <https://www.kaggle.com/nasa/astronaut-yearbook/data> and consists of 357 astronauts, with their demographic information. We ranked this

dataset by the number of space flight hours, and assigned individuals to categories based on their undergraduate major. A total of 83 majors are represented in the dataset, we assigned 9 most frequent - Physics (35), Aerospace Engineering (33), Mechanical Engineering (30) etc, to their individual categories, and combined the remaining 141 individuals into the category "other", resulting in $d = 10$ categories.

Pantheon. This dataset is a ranking of 11,341 individuals based on the popularity of their biographical page in Wikipedia, and is available at <http://pantheon.media.mit.edu/rankings/people/all/all/-4000/2010/H15>. Individuals in the dataset include historical and present-day figures, and are described with name, gender, birth year, place of birth, and occupation. Occupation is aggregated into a set of $d = 8$ cultural domains (<http://pantheon.media.mit.edu/methods>), which we use as the sensitive attribute to state diversity constraints.

Synthetic data. We also used synthetic data in our experiments, in cases where it was important to control dataset composition and assignments of scores to items in particular categories. Synthetic datasets consisted of three attributes: identifier of a tuple, value of the sensitive attribute and score. Additional details about specific datasets will be given as appropriate.

In our discussion, we will find it convenient to use the term *balanced* to describe datasets in which different categories are represented in the same proportion. For example, a dataset in which diversity is stated with respect to gender is balanced if about 50% of the individuals in the input are male and about 50% are female.

5.2 Diversity Constraints

Recall that our algorithms are designed for diversity constraints stated in terms of size limits on each category. The specific constraints chosen can implement different notions of diversity or fairness as discussed in Section 2. We explore several families of constraints, generated using the procedure described below, after a brief discussion of requirements on the constraints.

A single per-category constraint of the form $\text{floor}_i \leq k_i \leq \text{ceil}_i$ is satisfiable if $\text{floor}_i \leq n_i$, where n_i is the number of items in category i in the input. While it is not incorrect to set $\text{ceil}_i > n_i$, this will not lead to sensible subset selection in practice, and will make satisfiability of a set of constraints more cumbersome to state. For these reasons, we also require that $\text{ceil}_i \leq n_i$.

A set of per-category constraints is *satisfiable* if two conditions hold: $\sum_{i=1}^d \text{floor}_i \leq K$ and $\sum_{i=1}^d \text{ceil}_i \geq K$.

We use several measures of diversity, listed below, to generate a set of per-category constraints of the form $\text{floor}_i \leq k_i \leq \text{ceil}_i$ for a given selected set size K and number of categories d . We generate constraints that are satisfiable individually and as a set, as discussed above. In what follows, we assume that each category i is represented in the input dataset, that is, that $n_i \geq 1$.

Minimum: (Cover as many categories as possible.)

If $K \geq d$, set $\text{floor}_i = \text{ceil}_i = 1$ for all d categories. Next, compute $r = K - d$. If $r > 0$, assign the remaining r positions in the top- K to a random category j by setting $\text{ceil}_j = \text{ceil}_j + r$. Select category j from among categories in which $n_j \geq \text{ceil}_j + r$.

If $K < d$, assign $\text{floor}_i = \text{ceil}_i = 1$ to a random set of K out of d categories, and $\text{floor}_i = \text{ceil}_i = 0$ to the remaining $d - K$ categories.

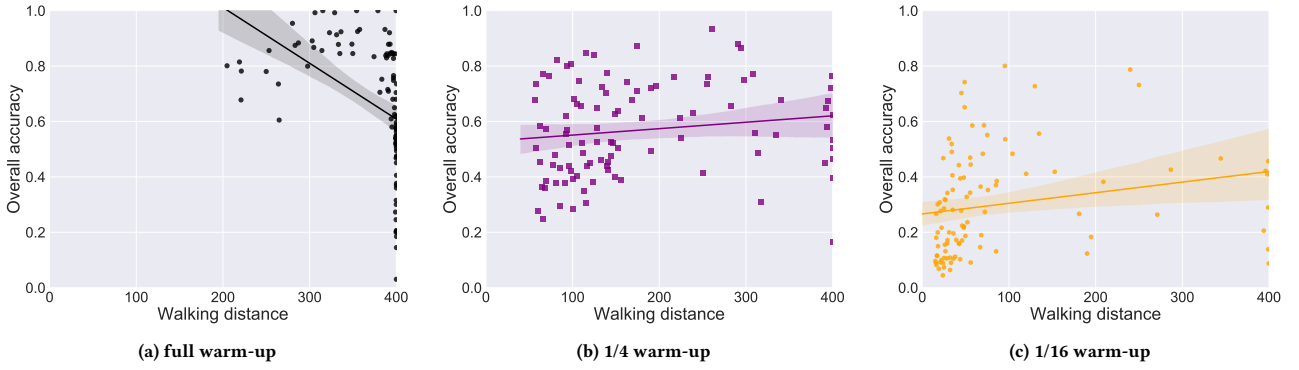


Figure 3: Accuracy of Algorithm 2 as function of walking distance, for different warm-up period lengths. Forbes US Richest, $K = 4$, $N = 400$ items, diversity on gender ($d = 2$), with average constraints $\text{floor}_i = \text{ceil}_i = K/d$.

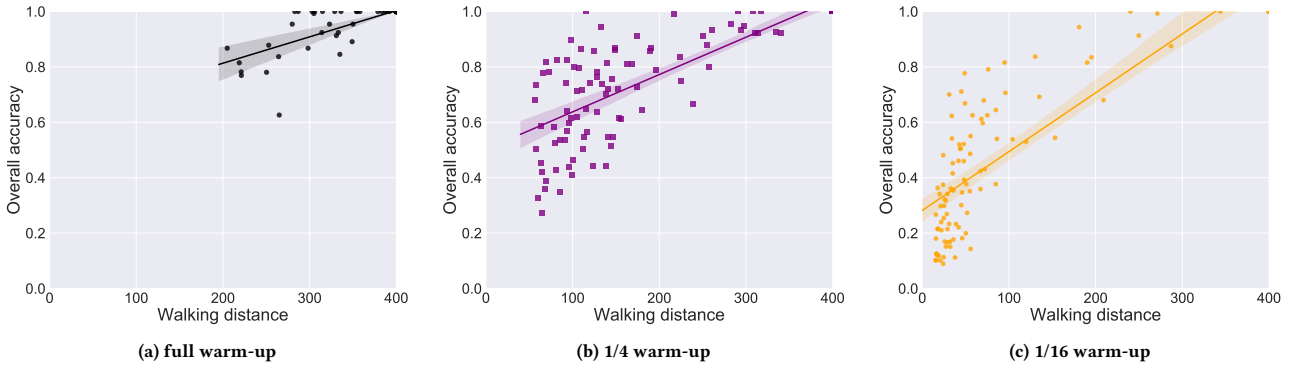


Figure 4: Accuracy of Algorithm 3 as function of walking distance, for different warm-up period lengths. Forbes US Richest, $K = 4$, $N = 400$ items, diversity on gender ($d = 2$), with average constraints $\text{floor}_i = \text{ceil}_i = K/d$.

Average: (Select equal numbers from each category.)

If $K \geq d$, set $\text{floor}_i = \text{MIN}(\lfloor K/d \rfloor, n_i)$ and $\text{ceil}_i = \text{MIN}(\lceil K/d \rceil, n_i)$ for all d categories. Next, compute $r = \sum_{i=1}^d \text{ceil}_i$. If $r < K$, assign the remaining r positions in the top- K to a random category j by setting $\text{ceil}_j = \text{ceil}_j + r$. Select category j from among categories in which $n_j \geq \text{ceil}_j + r$.

If $K < d$, set constraints as in **minimum** above.

Proportion: (Select equal proportions from each category.)

Recall that N denotes the size of the input.

If $K \geq d$, set $\text{floor}_i = \lfloor K * n_i / N \rfloor$ and $\text{ceil}_i = \lceil K * n_i / N \rceil$.

If $K < d$, set constraints as in **minimum** above.

Relaxed average: Let integer t denote the tightness threshold.

If $K \geq d$, set constraints as in **average** above. Next, set $\text{floor}_i = \text{MAX}(\text{floor}_i - t, 0)$ and $\text{ceil}_i = \text{MIN}(\text{ceil}_i + t, n_i)$.

If $K < d$, set constraints as in **minimum** above.

Relaxed proportion: Let integer t denote the tightness threshold.

If $K \geq d$, set constraints as in **proportion** above. Next, set $\text{floor}_i = \text{MAX}(\text{floor}_i - t, 0)$ and $\text{ceil}_i = \text{MIN}(\text{ceil}_i + t, n_i)$.

If $K < d$, set constraints as in **minimum** above.

Note that in balanced datasets, average and proportion constraints are equivalent, as are relaxed average and relaxed proportion (for the same tightness threshold t).

5.3 Metrics

Recall that Algorithms 2 and 3 both consume the input one item at a time, and decide whether to accept or reject an item when it is encountered. Algorithm 3 differs from Algorithm 2 in that it can place an item on the deferred list, and decide after it has considered all feasible items which of these to accept. For a given input (a fixed set of items received in some fixed order), Algorithms 2 and 3 will stop consuming the input at some point. We refer to this point — the number of items considered from the input stream, as the *walking distance*, and use it as our primary measure of efficiency. This measure is sometimes called *depth* in the top- k literature.

In several experiments with online algorithms, we quantify the relationship between algorithm efficiency and accuracy. To quantify accuracy, we use an intuitive normalized measure that compares the scores of the K retrieved items with the best possible K scores, subject to diversity constraints. Based on our statement of optimality in Theorem 3.2, we use scores returned by Algorithm 1 as the gold standard.

To make accuracy insensitive to shifts in the score distribution, we subtract the minimum observed score from each value. For example, suppose that the lowest net worth of any individual

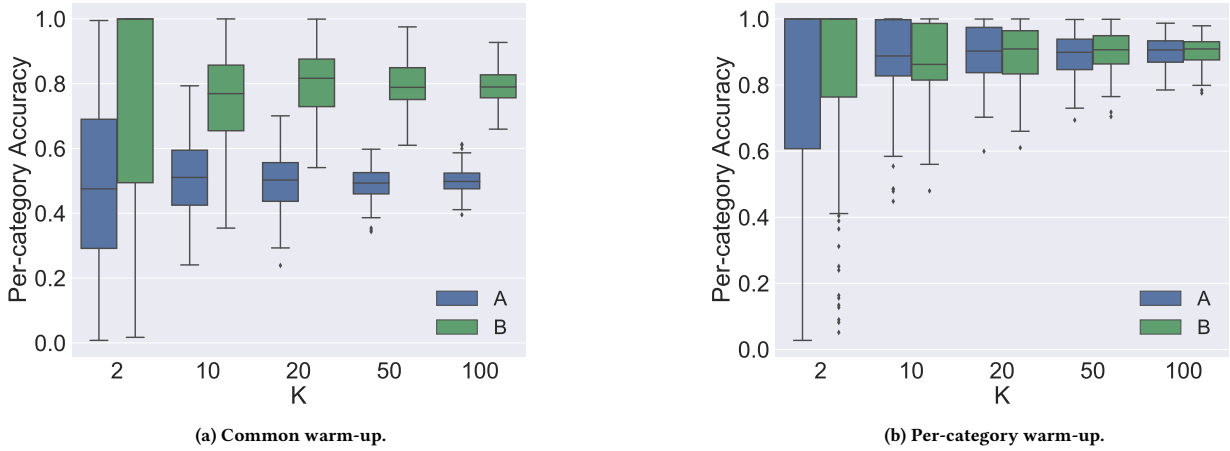


Figure 5: Per-category accuracy of Algorithm 2 as a function of K , on a balanced synthetic dataset of size $N = 10,000$ with average constraints. Category A items in the input have strictly lower scores than category B items. A common warm-up period leads to lower accuracy for both categories compared to per-category warm-up periods, and places items in category A at a particular disadvantage.

was 100, that $K = 2$ individuals were selected, with scores 225 and 200, and that the two highest-scoring individuals in the dataset, subject to diversity constraints, have scores 300 and 250, respectively. Then accuracy is computed as $\frac{(225-100)+(200-100)}{(300-100)+(250-100)}$.

5.4 Experimental Results: Online Algorithms

The most important question we are interested in is how well our streaming algorithms do despite being forced to meet set-oriented constraints while making decisions on individual items one at a time. The way the streaming algorithms are stated, they guarantee that the diversity constraints will be met, but do not guarantee optimality of utility score. We measure this in terms of accuracy, as described above.

The one parameter we can control in the streaming algorithms is the length of the warm-up period. Therefore, we start by presenting the relationship between warm-up period length and accuracy of the online algorithms of Section 4. To show this relationship, we will consider Figures 3 and 4, where 400 US Richest individuals (see Section 5.1 for dataset description) were randomly permuted, and where a diversity constraint was specified over the binary gender attribute, with $2 \leq k_F \leq 2$ and $2 \leq k_M \leq 2$ (a tight average constraint) and with $K = 4$. Each point in these figures corresponds to an execution of the relevant algorithm on a random permutation, with 100 executions per experiment (note that points may coincide). Other datasets in our experiments exhibited a similar trend.

We see that Algorithm 3 gets very high accuracy, often equaling the gold standard, if given enough warm up. Algorithm 2 also does not do too badly in terms of accuracy, though Algorithm 3 does substantially better.

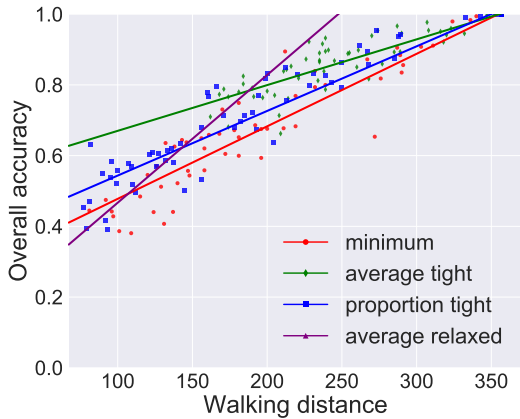
Walking distance takes on values between K (output size) and N (input size). Walking distance is made up of two parts: length of the warm-up period during which an on-line algorithm is estimating item scores, and post-warm-up, during which an algorithm is able to accept items. In both Algorithms 2 and 3, the per-category warm-up period has length $\lfloor \frac{N_i}{e} \rfloor$ for category i , while the total

warm-up period length is the sum of per-category warm-up period lengths, and of $\lfloor \frac{N}{e} \rfloor$ for the category-independent warm-up.

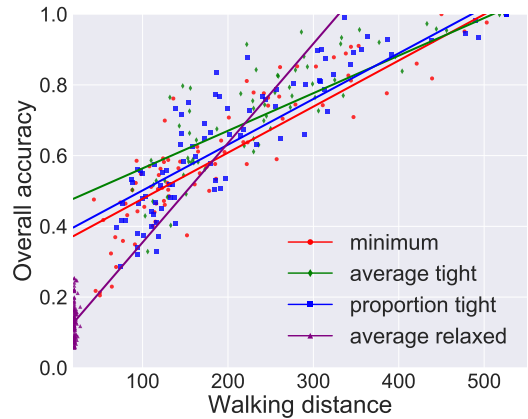
Consider Figures 3a and 4a, which show the overall accuracy as a function of walking distance for Algorithms 2 and 3, respectively. Note that accuracy of Algorithm 2 varies significantly at walking distance 400 – the case when all N items were consumed from the stream. This effect is so pronounced that in Figure 3a the over-all trend in accuracy is decreasing, due exclusively to this end-of-stream effect. In contrast, Algorithm 3 is not forced to accept items from the end of the stream, and so its accuracy strictly increases as a function of walking distance.

We do not have explicit control of the post-warm-up walking distance (because we must return a valid set of results – K results that meet the diversity constraints). However, we can impact walking distance by changing the length of the per-category warm-up periods, set to $\lfloor \frac{N_i}{e} \rfloor$ by default. These settings provide a strong theoretical guarantee, but can be conservative in practice, particularly for Algorithm 3 (the deferred list variant). Figures 3b and 3c show accuracy when warm-up is abbreviated to a quarter and a sixteenth of the optimal for Algorithm 2, and Figures 4b and 4c correspond to Algorithm 3. Observe that reducing warm-up period length introduces a trade-off between walking distance (efficiency) and accuracy, and that accuracy is often comparable to that which results from the full warm-up period, but at a lower efficiency cost.

In the next experiment we demonstrate the importance of having per-category warm-up periods. Recall that Algorithm 2 considers, and rejects, $\lfloor \frac{N_i}{e} \rfloor$ items in each category before accepting any items. This warm-up period allows the algorithm to form an expectation on the score of an item. Suppose now that $K = 2$, that there are two categories A and B in the input, and that diversity constraints are such that exactly one item per category is to be selected. Further, suppose that scores of A-items are strictly lower than scores of B-items. If a common (rather than a per-category) warm-up period were used by the algorithm, with a common MinHeap of score thresholds T , then



(a) NASA Astronauts, $N = 357$, $K = 30$, $d = 10$.



(b) Forbes World Richest, $N = 526$, $K = 20$, $d = 5$.

Figure 6: Accuracy of Algorithm 3 under different diversity constraints, at $1/8$ warm-up.

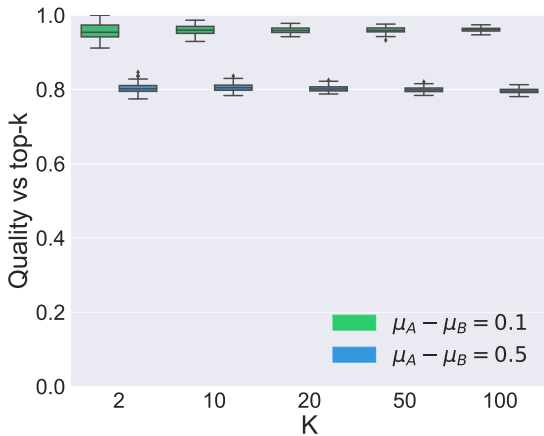


Figure 7: The cost of diversity: quality of the selected set is lower, in absolute terms, when items of a lower score must be included to satisfy diversity constraints. Synthetic datasets with $N = 10,000$, $d = 2$, under average constraints. Item scores are drawn from Gaussian distributions with the same standard deviation but different means (μ_A and μ_B).

T would contain the highest-scoring B-items that were encountered during warm-up. Since A-item scores are lower than B-item scores, no post-warm-up A-item will have a score that exceeds $\text{getMinElement}(T)$. This would force the algorithm to walk down the end of the stream in all cases (impacting performance), and to accept the very last A-item from the stream (impacting accuracy for category A).

To illustrate this point, we generate a synthetic dataset of $N = 10,000$ items in two categories, A and B, with a balanced breakdown — 5,000 A-items and 5,000 B-items, and with category-dependent scores. Scores of A-items are drawn uniformly at random from the $[0, 0.5)$ range, while scores of B-items are drawn uniformly at random from $[0.5, 1)$. We vary K between 2 and

100, and impose a (tight) average diversity constraint, setting $\lfloor \frac{K}{2} \rfloor \leq k_F \leq \lfloor \frac{K}{2} \rfloor$ and $\lfloor \frac{K}{2} \rfloor \leq k_M \leq \lfloor \frac{K}{2} \rfloor$.

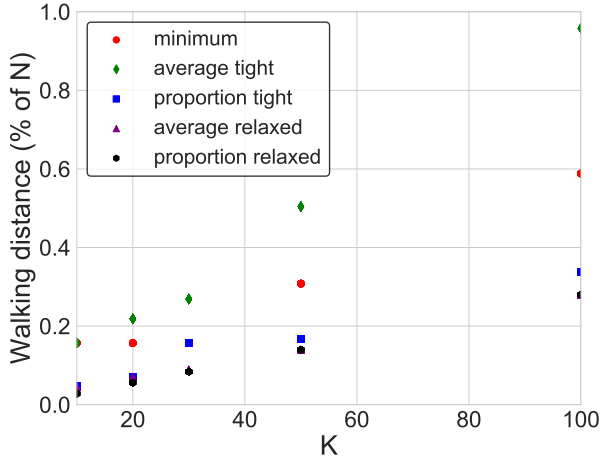
Figure 5 shows box-and-whiskers plots in which accuracy is presented as a function of K (see Section 5.3 for a description of how accuracy is computed). Observe that that accuracy for category A is lower than accuracy for category B in all cases when a common threshold is used (see Figure 5a). This is in contrast to the per-category threshold case in Figure 5b, where accuracy is comparable across the two categories.

Let us now compare performance of Algorithm 3 under different diversity constraints. Figure 6 presents accuracy as a function of walking distance, with warm-up period of length $\frac{1}{8}$ of the theoretically optimal (so, $\lfloor \frac{n_i}{8 * d} \rfloor$ per category), for two real datasets: NASA Astronauts ($N = 327$, $d = 10$ and $K = 30$) and the World’s Richest ($N = 526$, $d = 5$, $K = 20$). We also experimented with other real datasets, and with different values of K and warm-up period lengths, and present here results that are representative.

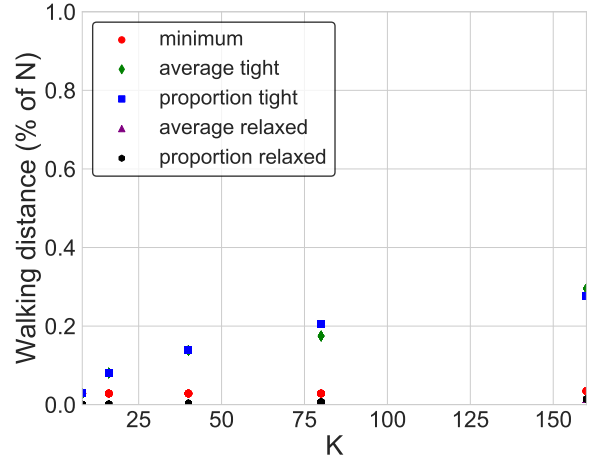
The average relaxed constraint (purple line in Figure 6) uses $t = \lfloor 0.3 * k \rfloor$ as the threshold (see Section 5.3 for a description of this and other constraints) and is easier to satisfy than the tight constraints, leading to somewhat lower walking distance. This difference was somewhat less pronounced in Figure 6a than in Figure 6b, and is sensitive to the variation in dataset composition (how balanced the categories are) and to the value of K .

In our final experiment with online algorithms, we quantified the impact of warm-up period length on the variance in accuracy. We generated a synthetic dataset with $N = 10,000$ items and with $d = 2$ categories. We requested that $K = 10$ items be returned by Algorithms 2 and 3, subject to proportion constraints.

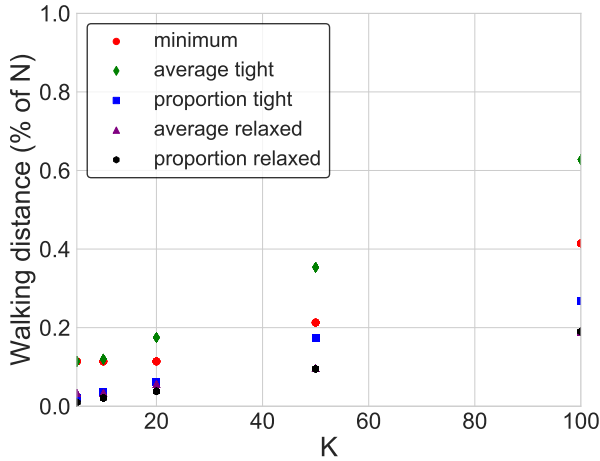
Scores of A-items were drawn uniformly at random from $[0, 0.5)$, while scores of B-items were drawn from $[0.5, 1)$. (Note that we executed per-category warm-up in both algorithms, and so differences in scores between A and B do not impact accuracy, as we saw in Figure 5.) We generated three such datasets, with A constituting 10%, 25% and 50% of the over-all dataset. For Algorithm 2, we observed, as expected, that higher variance in accuracy occurs when A appears in the dataset in lower proportion: variance was 0.080 for 10% proportion, 0.075 for 25% proportion and 0.019 for 50% proportion. Variance did not differ significantly for Algorithm 3.



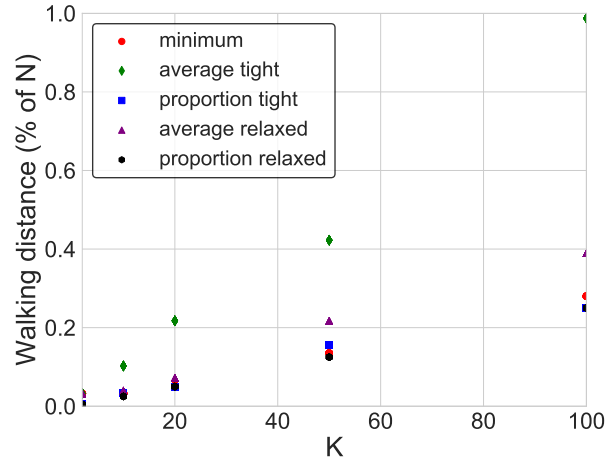
(a) NASA Astronauts, $N = 357$, $d = 10$.



(b) Pantheon, $N = 11,341$, $d = 8$.



(c) Forbes World's Richest, $N = 526$, $d = 5$.



(d) Forbes US Richest, $N = 400$, $d = 2$.

Figure 8: Walking distance of Algorithm 1 in proportion to N , as a function of K , for different diversity constraints.

5.5 Experimental Results: Static Algorithm

In this experiment we quantify the cost of imposing diversity constraints, in terms of a normalized measure we call *quality*: the sum of score of the diverse top- K divided by the sum of scores of the “vanilla” (category-agnostic) top- K . Figure 7 presents box-and-whiskers plots that quantify this cost of diversity as a function of K for two synthetic datasets, each of size N , with $d = 2$ categories represented in equal proportion, and with average diversity constraint. In the both datasets, item scores were drawn from per-category Gaussian distributions, with different means but the same standard deviations. The dataset presented in green had means of A-scores and of B-scores close to each other ($\mu_A - \mu_B = 0.1$), while in the dataset presented in blue, these distributions were further apart ($\mu_A - \mu_B = 0.5$). As expected, the cost of diversity is higher in the latter case, since items of lower scores (in absolute terms) must be included into the result to satisfy diversity constraints.

In our final experiment we use real datasets to support the claim that, while Algorithm 1 may walk to the end of the input in the worst case, this rarely happens in practice. Figure 8 presents walking distance of Algorithm 1 as a function of K for different diversity constraints, for the NASA Astronauts, Pantheon, Forbes World's Richest, and Forbes US Richest datasets. A value of 1 on the y -axis denotes that the algorithm walked to the end of the list (that is, walking distance equals N). We observe that this does not occur often, particularly for lower values of K .

6 RELATED WORK

There is considerable work on diverse top- k , starting with [2, 16], see also [18] (Sections 5.1, 5.6 and 6) for a survey. The work proposed here differs from prior work in that we consider a family of diversity constraints that can express coverage-based (rather than distance-based) diversity [7], and can also be used to compute

several fairness metrics — those based on proportional representation. To the best of our knowledge, diversity in combination with utility has not been considered in a fully online setting.

In [16] a generic method is proposed to extend top- k algorithms with a diversity criterion that is based on pair-wise similarity. The problem is formulated as: given a user-defined pair-wise similarity function $sim(s_i, s_j)$, and a user-defined similarity threshold τ , return the highest-scoring set of k items such that $sim(s_i, s_j) \leq \tau$. When describing top- k methods, they refer to incremental methods (generate results in decreasing order of scores, stop once k results were generated) vs. bounding methods (generate results in some order, stop once the top- k are among the results, Fagin’s TA is in this category).

More recently, diversity-aware top- k for pub/sub queries over text streams was considered in [6]. There, diversity is a pair-wise measure based on document similarity in the top- k (max-sum), quality is measured as the relevance of a document to a user’s query, and there is additionally a per-document recency score that is appropriate for a stream of Twitter messages or Facebook status updates, and is based on an exponential decay function.

Another related line of work is [1], where max-sum diversity is maximized subject to a constraint on a variant of coverage-based diversity. The problem is posed as a partition matroid, a local search algorithm is proposed, and it is shown that it achieves a 0.5 approximation of the optimal solution.

Selection of diverse set results in an online setting is studied in [15]. The authors consider selection of subsets of items that are simultaneously diverse along multiple dimensions. For example, for program committee selection it is desirable to achieve coverage of topics, geographic diversity and gender diversity. The paper proposes and analyzes several diversity objectives and proposes heuristic and dynamic programming methods. The most important difference between our method and that of [15] is that they do not explicitly handle utility.

In [16] a set of diverse top- k items is determined that maximizes the total score of the K selected items, subject to a pair-wise diversity constraint. The problem is modeled by representing the items as vertices in a graph, and by including an edge between vertices s_i and s_j if their similarity is above a user-specified threshold τ . A diverse set K is the independent set of a graph — a set of vertices in which no two vertices are adjacent. This formulation can accommodate the coverage diversity version of our problem: Include an edge between two vertices if they belong to the same category, then identify an independent subset of size d that maximizes the total score. If $d < k$, we add $k - d$ vertices that maximize the total score. The algorithmic contributions of [16] are in (1) determining when the set of $k' > k$ items is sufficient to compute the true diverse top- k , and (2) efficiently identifying the independent set of a graph for a fixed k (finding an independent set of a graph is NP-hard).

7 CONCLUSIONS

Diversity and group fairness are important objectives in algorithmic decision-making. Since most algorithms are designed to score or classify items individually, it is not easy to support these objectives. In this paper, we showed how we can continue to select items individually and meet desired diversity and group fairness constraints, while paying a very small utility cost.

We demonstrated experimentally that the theoretically-motivated setting for warm-up period length can be conservative in practice.

In our future work, we will investigate the interaction between expected (or observed) score variance and warm-up period length.

Further, we demonstrated that different categories must be treated separately in score estimation, to achieve comparable accuracy irrespective of expected score, and ultimately afford comparable opportunity to members of different groups.

REFERENCES

- [1] Zeinab Abbassi, Vahab S. Mirrokni, and Mayur Thakur. 2013. Diversity maximization under matroid constraints. In *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2013, Chicago, IL, USA, August 11-14, 2013*. 32–40. DOI: <http://dx.doi.org/10.1145/2487575.2487636>
- [2] Albert Angel and Nick Koudas. 2011. Efficient diversity-aware search. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*. 781–792. DOI: <http://dx.doi.org/10.1145/1989323.1989405>
- [3] Moshe Babaioff, Nicole Immorlica, David Kempe, and Robert Kleinberg. 2008. Online auctions and generalized secretary problems. *SIGecom Exchanges* 7, 2 (2008). DOI: <http://dx.doi.org/10.1145/1399589.1399596>
- [4] Moshe Babaioff, Nicole Immorlica, and Robert Kleinberg. 2007. Matroids, secretary problems, and online mechanisms. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7-9, 2007*. 434–443. <http://dl.acm.org/citation.cfm?id=1283383.1283429>
- [5] Solon Barocas and Andrew D. Selbst. 2016. Big data’s disparate impact. *California Law Review* 104 (2016).
- [6] Lisi Chen and Gao Cong. 2015. Diversity-Aware Top-k Publish/Subscribe for Text Stream. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. 347–362. DOI: <http://dx.doi.org/10.1145/2723372.2749451>
- [7] Marina Drosou, HV Jagadish, Evaggelia Pitoura, and Julia Stoyanovich. 2017. Diversity in Big Data: A Review. *Big Data* 5, 2 (2017).
- [8] E.B. Dynkin. 1963. The optimum choice of the instant for stopping a Markov process. *Sov. Math. Dokl.* 4 (1963).
- [9] Ronald Fagin, Amnon Lotem, and Moni Naor. 2001. Optimal Aggregation Algorithms for Middleware. In *Proceedings of the Twentieth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 21-23, 2001, Santa Barbara, California, USA*. DOI: <http://dx.doi.org/10.1145/375551.375567>
- [10] Michael Feldman, Sorelle A. Friedler, John Moeller, Carlos Scheidegger, and Suresh Venkatasubramanian. 2015. Certifying and Removing Disparate Impact. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Sydney, NSW, Australia, August 10-13, 2015*. 259–268. DOI: <http://dx.doi.org/10.1145/2783258.2783311>
- [11] Thomas S. Ferguson. 1989. Who Solved the Secretary Problem? *Statist. Sci.* 4, 3 (08 1989), 282–289. DOI: <http://dx.doi.org/10.1214/ss/1177012493>
- [12] Sorelle A. Friedler, Carlos Scheidegger, and Suresh Venkatasubramanian. 2016. On the (im)possibility of fairness. *CoRR abs/1609.07236* (2016). <http://arxiv.org/abs/1609.07236>
- [13] Ravi Kumar, Silvio Lattanzi, Sergei Vassilvitskii, and Andrea Vattani. 2011. Hiring a secretary from a poset. In *Proceedings 12th ACM Conference on Electronic Commerce (EC-2011), San Jose, CA, USA, June 5-9, 2011*. 39–48. DOI: <http://dx.doi.org/10.1145/1993574.1993582>
- [14] D. V. Lindley. 1961. Dynamic Programming and Decision Theory. *Journal of the Royal Statistical Society* 10, 1 (March 1961), 39–51.
- [15] Debmalya Panigrahi, Atish Das Sarma, Gagan Aggarwal, and Andrew Tomkins. 2012. Online selection of diverse results. In *Proceedings of the Fifth International Conference on Web Search and Web Data Mining, WSDM 2012, Seattle, WA, USA, February 8-12, 2012*. 263–272. DOI: <http://dx.doi.org/10.1145/2124295.2124329>
- [16] Lu Qin, Jeffrey Xu Yu, and Lijun Chang. 2012. Diversifying Top-K Results. *PVLDB* 5, 11 (2012), 1124–1135. http://vldb.org/pvldb/vol5/p1124_luqin_vldb2012.pdf
- [17] Julia Stoyanovich, Bill Howe, Serge Abiteboul, Gerome Miklau, Arnaud Sahuguet, and Gerhard Weikum. 2017. Fides: Towards a Platform for Responsible Data Science. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management, Chicago, IL, USA, June 27-29, 2017*. 26:1–26:6. DOI: <http://dx.doi.org/10.1145/3085504.3085530>
- [18] Kaiping Zheng, Hongzhi Wang, Zhixin Qi, Jianzhong Li, and Hong Gao. 2017. A survey of query result diversification. *Knowl. Inf. Syst.* 51, 1 (2017), 1–36. DOI: <http://dx.doi.org/10.1007/s10115-016-0990-4>
- [19] Indre Zliobaite. 2017. Measuring discrimination in algorithmic decision making. *Data Min. Knowl. Discov.* 31, 4 (2017), 1060–1089. DOI: <http://dx.doi.org/10.1007/s10618-017-0506-1>

Apollo: Learning Query Correlations for Predictive Caching in Geo-Distributed Systems

Brad Glasbergen
University of Waterloo
bjglasbe@uwaterloo.ca

Michael Abebe
University of Waterloo
mtabebe@uwaterloo.ca

Khuzaima Daudjee
University of Waterloo
kdaudjee@uwaterloo.ca

Scott Foggo
University of Waterloo
sjfoggo@uwaterloo.ca

Anil Pacaci
University of Waterloo
apacaci@uwaterloo.ca

ABSTRACT

The performance of modern geo-distributed database applications is increasingly dependent on remote access latencies. Systems that cache query results to bring data closer to clients are gaining popularity but they do not dynamically learn and exploit access patterns in client workloads. We present a novel prediction framework that identifies and makes use of workload characteristics obtained from data access patterns to exploit query relationships within an application’s database workload. We have designed and implemented this framework as Apollo, a system that learns query patterns and adaptively uses them to predict future queries and cache their results. Through extensive experimentation with two different benchmarks, we show that Apollo provides significant performance gains over popular caching solutions through reduced query response time. Our experiments demonstrate Apollo’s robustness to workload changes and its scalability as a predictive cache for geo-distributed database applications.

1 INTRODUCTION

Modern distributed database systems and applications frequently have to handle large query processing latencies resulting from the geo-distribution of data [11, 13, 41]. Industry reports indicate that even small increases in client latency can result in significant drops in both web traffic [20] and sales [3, 30]. A common solution to this latency problem is to place data closer to clients [38, 39] using caches, thereby avoiding costly remote round-trips to datacenters [27]. Static data, such as images and video content, is often cached on servers geographically close to clients. These caching servers, called *edge nodes*, are a crucial component in industry architectures. To illustrate this, consider Google’s datacenter and edge node locations in Figure 1. Google has comparatively few datacenter locations relative to edge nodes, and the latency between the edge nodes and datacenters can be quite large. Efficiently caching data on these edge nodes substantially reduces request latency for clients.

Existing caching solutions for edge nodes and content delivery networks (CDN) focus largely on static data, necessitating costly round trips to remote data centers for requests relying on dynamic data [21]. Since a majority of webpages today are generated dynamically [5], a large number of requests are not satisfied by cached data, thereby incurring significant latency penalties. We address this concern in Apollo, a system that exploits client access patterns to intelligently prefetch and cache dynamic data on edge nodes.

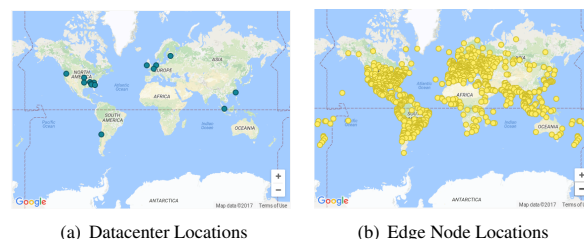


Figure 1: Google’s datacenter and edge node locations [21].

```

1. SELECT C_ID FROM CUSTOMER WHERE
   C_UNAME = @C_UN and C_PASSWD = @C_PAS

2. SELECT MAX(O_ID) FROM ORDERS WHERE
   O_C_ID = @C_ID

3. SELECT ... FROM ORDER_LINE, ITEM
   WHERE OL_I_ID = I_ID and OL_O_ID = @O_ID

```

Figure 2: A set of motivating queries in TPC-W’s Order Display web interaction. Boxes of the same colour indicate shared values across queries.

Database client workloads often exhibit query patterns, corresponding to application usage patterns. In many workloads [1, 10, 42], queries are highly correlated. That is, the execution of one query determines which query executes next and with what parameters. These dependencies provide opportunities for optimization through predictively caching queries. In this paper, we focus on discovering relationships among queries in a workload. We exploit the discovered relationships to predictively execute future dependent queries. Our focus is to reduce the response time of consequent queries by predicting and executing them, caching query results ahead of time. In doing so, clients can avoid contacting a database located at a distant datacenter, satisfying queries instead from the cache on a closer edge node.

As examples of query patterns, we consider a set of queries from the TPC-W benchmark [42]. In this benchmark’s Order Display web interaction, shown in Figure 2, we observe that the second query is dependent upon the result set of the first query. Therefore, given the result set of the first query, we can determine the input set of the second query, *predictively execute* it, and *cache* its results. After the second query has executed, we can use its result set as input to the third query, again presenting an

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

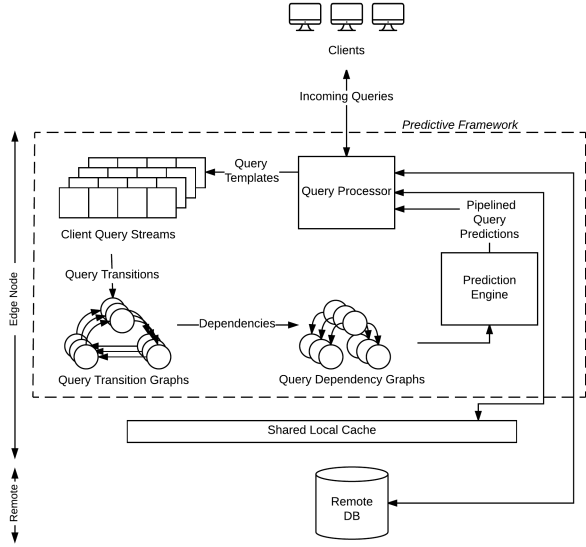


Figure 3: Query flow through components of the predictive framework.

opportunity for predictive caching. Similar scenarios abound in the TPC-W and TPC-C benchmarks, such as in the TPC-W Best-Seller web interaction and in the TPC-C Stock level transaction. Examples that benefit from such optimization, including real-world applications, have been previously described [10].

In this paper, we propose a novel prediction framework that uses a query-pattern aware technique to improve performance in geo-distributed database systems through caching. We implement this framework in Apollo, which uses a query transition graph to learn correlations between queries and to predict future queries. In doing so, Apollo determines query results that should be cached ahead of time so that future queries can be satisfied from a cache deployed close to clients. Apollo prioritizes common and expensive queries for caching, eliminating or reducing costly round-trips to remote data without requiring modifications to the underlying database architecture. Apollo’s ability to learn allows it to rapidly adapt to workloads in an online fashion. Apollo is designed to enhance an existing caching layer, providing predictive caching capabilities for improved performance.

The contributions of this paper are threefold:

- (1) We propose a novel *predictive* framework to identify relationships among queries and predict consequent ones. Our framework uses *online learning* to adapt to changing workloads and reduce query response times (Section 2).
- (2) We design and implement our framework in a system called Apollo, which predictively executes and caches query results on edge nodes close to the client (Section 3).
- (3) We deploy and extensively test Apollo on Amazon EC2 using the TPC-W and TPC-C benchmark workloads to show that significant performance gains can be achieved for different query workloads (Section 4).

2 PREDICTING QUERIES

A client’s database workload is comprised of a stream of queries and the transitions between them. These queries are synthesized into the *query transition graph*, which is at the core of our predictive framework. From this query transition graph, we discover

query relationships, dependencies and workload characteristics for use in our predictive framework. The predictive framework stores query result sets in a shared local cache, querying the remote database if a client submits a query for which the cache does not have the results.

Figure 3 gives a high level overview of how incoming queries are executed, synthesized into the query transition graph, and used for future query predictions. Incoming queries are routed to the query processor, which retrieves query results from a shared local query result cache, falling back to a remote database on a cache miss. Query results are immediately returned to the client and, together with their source queries, are mapped to more generalized *query template* representations (Section 2.1). These query templates are placed into per-client queues of queries called query streams, which are continuously scanned for relationships among executed queries. Query relationships are synthesized into the query transition graph and then used to detect query correlations, discovering dependencies among executed queries and storing them in a dependency graph. This dependency graph is used by the prediction engine to predict consequent queries given client queries that have executed.

Although we focus on geographically distributed edge nodes with remote datacenters, Apollo can also be deployed locally as a middleware cache. Our experiments in Section 4 show that both deployment environments benefit significantly from Apollo’s predictive caching.

Next, we discuss the abstractions and algorithms of our predictive framework, describing how queries flowing through the system are merged into the underlying models and used to predict future queries.

2.1 Query Templates

Using a transition graph to reason about query relationships requires a mapping from database workloads (queries and query relationships) to transition structures (query templates and template transitions). We propose a formalization of this mapping through precise definitions, and then show how our model can be used to predict future queries.

Queries within a workload are often correlated directly through *parameter sharing*. Motivated by the Stock Level transaction in the TPC-C benchmark, consider an example of parameter sharing in which an application executes query Q_1 to look up a product ID followed by query Q_2 to check the stock level of a given product ID. A common usage pattern is to execute Q_1 , and then use the returned product ID as an input to Q_2 to check that product’s stock level. In this case, Q_2 is directly related to Q_1 via a *dependency* relationship. Specifically, Q_2 relies on the output of Q_1 to execute.

We generalize our model by tracking relationships among query templates rather than among parameterized queries. Two queries Q_1 and Q_2 have the same query template if they share the same statement text barring constants that could logically be replaced by placeholders for parameters values (“?”). Each query template is represented by a node in the query transition graph.

Below is an example of two queries (Q_1, Q_1') and their corresponding templates (Qt_1, Qt_1'):

```

 $Q_1$ : SELECT C_ID FROM CUSTOMER WHERE C_UNAME = 'Bob' and C_PASSWD = 'pwd'
 $Qt_1$ : SELECT C_ID FROM CUSTOMER WHERE C_UNAME = ? and C_PASSWD = ?

```

```

Q1: SELECT C_ID FROM CUSTOMER WHERE C_UNAME =
'Alice' and C_PASSWD = 'pwd2'
Q1': SELECT C_ID FROM CUSTOMER WHERE C_UNAME =
? and C_PASSWD = ?

```

Note that although the above two original queries differ, their query templates are the same. Therefore, a node's transitions in the transition graph are based on query relationships from both Q_1 and Q_1' .

2.2 Query Template Relationships

To find query template relationships, we implement the transition graph as a frequency-based Markov graph, constructing it in an online fashion. We exploit the memory-less property of Markov models to simplify transition probability computations — transition probabilities are based solely on the previous query the client executed.

We monitor incoming queries, map them to query templates and calculate template transition probabilities. In particular, for any two templates Q_{t_i}, Q_{t_j} , we create an edge from Q_{t_i} to Q_{t_j} if Q_{t_j} is executed after Q_{t_i} . We store the probability of Q_{t_j} executing after Q_{t_i} on this edge, and refer to it as $P(Q_{t_j}|Q_{t_i})$. If this probability is larger than some configurable threshold τ , we say Q_{t_j} is related to Q_{t_i} .

The τ parameter serves as a configurable confidence threshold for query template relationships. More concretely, the τ parameter provides the minimum required probability for Q_{t_j} executing after Q_{t_i} to infer that they are related. By choosing τ appropriately, we can limit the predictive queries executed after seeing Q_{t_i} to only those that are highly correlated to it. In doing so, we ensure that our predictions have a high degree of accuracy and avoid inundating the database with predictive executions of unpopular queries.

$P(Q_{t_j}|Q_{t_i})$ is too broad to capture fine-grained query template relationships. Given enough time, almost all of the query templates in a workload could be considered related under the above definition. Two templates should not be considered related if there is a significant time gap between them, thus motivating a temporal constraint. Furthermore, by placing a temporal restriction on the relationship property, we reduce the time needed to look for incoming related templates. Consequently, we define a configurable duration, Δt , which specifies the maximum allowable time separation between related query templates.

Definition 2.1. For any two query templates Q_{t_i}, Q_{t_j} , in which Q_{t_j} is executed T time units apart from Q_{t_i} , if $P(Q_{t_j}|Q_{t_i}; T \leq \Delta t) > \tau$ for some threshold parameter $\tau \in [0, 1]$, we consider Q_{t_j} to be a related query template of Q_{t_i} .

To learn a transition graph representing $P(Q_{t_j}|Q_{t_i}; T \leq \Delta t)$, we map executed queries to query templates and place them at the tail of per-client queues called query streams. Since each client has its own stream and transition graph, we avoid expensive lock contention when updating the graphs and computing transition probabilities.

Algorithm 1 runs continuously over client query streams, updating their corresponding transition graphs. Intuitively, the algorithm scans the query stream, looking for other query templates that executed within Δt of the first query template, adding counts to their corresponding edges and afterwards incrementing the vertex count indicating number of times the template has been seen. To calculate the probability of $P(Q_{t_j}|Q_{t_i}; T \leq \Delta t)$, we take the edge count from Q_{t_i} to Q_{t_j} and divide by the vertex count for Q_{t_i} . To use

Algorithm 1 Query Transition Graph Construction

Input: $(Q_{t_1}, t_1), (Q_{t_2}, t_2), \dots$, an infinite stream of incoming query template identifiers and their execution timestamps,
 Δt , a fixed time duration,
 $G = (V, E)$, a directed graph, initially empty,
 $w_v : V \rightarrow \mathbb{N}$, vertex counters indicating the number of times we have seen the vertex, initially all zero,
 $w_e : V \times V \rightarrow \mathbb{N}$, edge counters indicating the number of times we've seen the outgoing vertex followed by the incoming vertex within Δt , initially all zero.

```

i ← 1
loop
  if ti + Δt > now() then
    wait until now() > ti + Δt
  end if
  V ← V ∪ {Qti}
  wv(Qti) ← wv(Qti) + 1
  j ← i + 1
  loop
    if tj > ti + Δt then
      // too far apart in time
      break
    else
      E ← E ∪ {(Qti, Qtj)}
      we(Qti, Qtj) ← we(Qti, Qtj) + 1
    end if
    j ← j + 1
  end loop
  // advance forward in stream
  i ← i + 1
end loop

```

the variables directly from Algorithm 1, the probability that query template Q_{t_j} executes within Δt of a query template Q_{t_i} is given by $\frac{w_e(Q_{t_i}, Q_{t_j})}{w_v(Q_{t_i})}$. Per Definition 2.1, if this probability exceeds τ then Q_{t_j} is considered related to Q_{t_i} .

The choice of the Δt parameter can impact prediction efficacy. If Δt is too high, it is possible that relationships will be predicted where there are none; if Δt is too low, we may not discover relationships where they are present. Although the choice of Δt is workload dependent, some indicators aid us in choosing an appropriate value, such as query arrival rate. If $P(Q_{t_j}|Q_{t_i}; T \leq \Delta t)$ is high for a fixed Q_{t_i} and many different Q_{t_j} , then either Q_{t_i} is a common query template with many quick-executing related query templates, or Δt is set too high. If this holds for many different Q_{t_i} , then Δt can be decreased. A similar argument holds for increasing Δt . We discuss selection of Δt and τ values for various workloads in Section 4.7.

A key property of our model is that it uses online learning to adapt to changing workloads. As new query templates are observed, query template execution frequencies change, or query relationships adjust, the transition graph adapts to learn the changed workload. Moreover, online learning precludes the need to undergo expensive offline training before deployment. Instead, our model rapidly learns client workloads and takes action immediately.

2.3 Parameter Mappings

Predictive query execution requires a stronger relationship between queries than the transition graph provides. In addition to queries being related, they must also exhibit a dependency relationship.

To provide predictive execution capabilities, we record the output sets of query templates and match them with the input sets of templates that we have determined are related based on the transition graph. We then confirm each output column to input variable mapping over a verification period, after which only the mappings present in every execution are returned.

As a concrete example, consider the TPC-W queries from Figure 2. We will refer to the query template for the first query as Qt_1 and the template for the second query as Qt_2 . In the first stage of tracking, we observe which query templates have executed within Δt of Qt_1 . Once Qt_1 has executed enough times (according to the verification period), we begin to construct mappings among the query templates. After Qt_1 finishes an execution, we record its output set. When any of Qt_1 's related query templates (in this case assume only Qt_2) are executed, we record their input sets. We then check if any column's result in Qt_1 's output set maps to the input parameters of Qt_2 . If they do, we record the matching output columns with their corresponding input argument position. If the *same* mappings are observed across the verification period, we infer that these mappings always hold.¹ If a query template has mappings for every one of its input arguments from a set of prior templates, we can predict a query by forwarding parameters from its prior template's result sets as soon as they are available. In this case, we say the query template is a candidate for predictive execution given its prior query templates' result sets. Similarly, we discover mappings between Qt_2 and Qt_3 and use them to execute Qt_3 given Qt_2 's result set.

2.4 Pipelining Query Predictions

Parameter mappings among query templates enable predictive execution of queries as soon as their input sets are available via prior template execution. It may be the case that the prior query templates are also predictable, forming a hierarchical tree of dependencies among templates. We exploit these relationships by *pipelining* query predictions. Pipelining uses result sets from predictively executed queries as input parameters for future predictions, thereby enabling predictions several steps in advance.

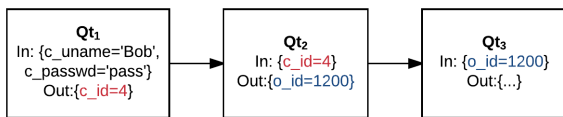


Figure 4: An example of pipelining within a dependency hierarchy. The arrows represent a mapping from a prior query template's output set to the consequent query template's input set.

Figure 4 illustrates how pipelining can be used to form extended chains of predictive executions using the TPC-W example from Figure 2. Qt_1 has a mapping to Qt_2 , which in turn has a mapping

¹If future executions disprove a mapping, we will mark that mapping invalid and preclude the template from predictive execution if its dependencies are no longer met.

to Qt_3 . If Qt_1 is executed, we can forward its result set as input with which to predictively execute Qt_2 . Once Qt_2 has also been executed, we can predictively execute Qt_3 . As such, Qt_2 is *fully defined* given the result set of Qt_1 , and Qt_3 is fully defined given the result set of Qt_2 . We formalize the notion of fully defined queries:

Definition 2.2. A fully defined query template (FDQ) Qt_j has all of its inputs provided by some set, possibly empty, of prior query templates $Qt_{i_1}, Qt_{i_2}, \dots, Qt_{i_k}$ where each Qt_{i_m} ($\forall m \in [1, k]$) is either:

- (1) a fully defined query template, or
- (2) a dependency query template, required to execute Qt_j .

Per Definition 2.2, both Qt_2 and Qt_3 are FDQs, but Qt_1 is simply a dependency query. This definition captures the dependency-graph nature of FDQs — each node in this graph corresponds to a query template, with inbound and outbound edges corresponding to inbound and outbound parameter mappings, respectively. The transition graph induces the dependency graph but is stored and tracked separately. By keeping the dependency graph separate, we reduce contention on it. Once the dependency graph matches the current workload, it will not need to be modified until the workload changes.

Algorithm 2 Core Prediction Algorithm

```

Input: executed query template  $Qt$ 
  record_query_template( $Qt$ )
  new_fdqs = find_new_fdqs( $Qt$ )
  rdy_fdqs = mark_ready_dependency( $Qt$ )
  rdy_fdqs = rdy_fdqs  $\cup$  new_fdqs
  ordered_fdqs = find_all_runnable_fdqs(rdy_fdqs)
  for all rdy_fdq  $\in$  ordered_fdqs do
    execute_fdq(rdy_fdq)
  end for
  
```

Discovering new FDQs, managing FDQ dependencies, and pipelining predictions comprise the main routine of the predictive framework. The engine executes Algorithm 2 after the execution of a client-provided instance of query template Qt . The engine records Qt 's result set and input parameters in the query transition graph (Section 2.3), looks for parameter mappings, and records discovered dependencies in the dependency graph. This query template is then marked as executed so that FDQ pipelines that depend on its result set can proceed. Any queries that are determined ready for execution given the result of this query (and previously executed queries) are then executed, forwarding parameters from their dependent queries' result sets. The dependencies are then reset, waiting for future invocations with which to predict queries. The dependency graph is stored as a hash map with edges between dependent queries, allowing Apollo to quickly determine which FDQs are ready for execution given an executed query.

Always defined query templates (ADQs) are a subset of FDQs, requiring that all of their prior query templates (recursively) are FDQs. They comprise an important subclass of fully defined queries since their dependencies are *always* satisfied; they can be executed and cached at any time. As a concrete example, "SELECT COUNT(*) FROM shopping_cart" is an ADQ because all of its input parameters (the empty set) are always satisfied.

It follows from Definition 2.2 that an FDQ is an ADQ if and only if all of its inputs are provided by ADQs. Consequently, ADQ

hierarchies are discovered by recursively checking the dependency structure of the FDQ.

3 APOLLO

In this section, we present Apollo, our system that implements the predictive framework described in Section 2. Apollo is a system layer placed between a client application and the database server. Application clients submit queries to the Apollo system, which then interacts with the database system and cache to return query results.

Apollo uses Memcached [19], a popular industrial-strength distributed caching system, as the query result cache. Each executed read-only query has its result set placed in Memcached, which employs the popular Least Recently Used (LRU) eviction policy. With predictive caching enabled, Apollo also places predictively executed query results into the cache, increasing the number of cache hits and thereby overall system performance. Apollo’s predictive engine operates in a *complementary* manner where queries are passed unchanged through to the cache and database, preserving the effective workload behaviour. Apollo executes predicted queries and caches them ahead of time, reducing response times through correlated query result caching.

Since Apollo is implemented in the Java programming language, we use the JDBC API to submit queries to the remote MySQL [33] database. The JDBC API [32] makes Apollo database agnostic and therefore portable, allowing MySQL to be easily swapped for any other JDBC compliant relational database system.

To efficiently track query templates within Apollo, we identify queries based on a hash of their constant independent parse tree. A background thread processes the SQL query strings placed into the query stream, parsing and then hashing them into a 64-bit identifier. All parameterizable constants are replaced by a fixed string, and therefore share the same hash code. Thus, queries with the same text modulo parameterizable constants have the same hash.

Hashes can be computed efficiently and are used internally to refer to query templates. Apollo uses them to look up nodes in the transition graph, and to find statistics and parameters we have stored for each query template. Hash collisions are very rare due to the length of the hash and common structures that SQL statements share. Due to the complementary nature of Apollo, query template hash collisions are guaranteed not to introduce incorrect system behaviour.

3.1 Prediction Engine

When a client submits a query, it has its results retrieved from the local cache or executed against the remote database, then placed into Apollo’s *query stream* and evaluated by the prediction engine. Background threads use the query stream to construct the transition graph described in Section 2, processing executed queries into query templates. The core prediction routine from Section 2.4 is then invoked: new FDQs are discovered from the underlying transition graph, the dependency graph is updated, and future queries are predicted using pipelining. We now detail each of these subroutines, showing how these operations are carried out efficiently.

Algorithm 3 shows how new FDQs are discovered. First, the transition graph is consulted for all related query templates (templates with inbound edges from Qt_i) since these are the templates that may have new mappings from Qt_i ’s result set. Qt_i itself is also

checked since it may be an ADQ (if it has no input parameters). For each query template Qt_j that has no recorded dependency information in the dependency graph, the transition graph is checked to see which templates have mappings to them. If each of Qt_j ’s input parameters are satisfied by its prior query templates then by Definition 2.2 we know that it is an FDQ. An FDQ structure is constructed for Qt_j and its dependencies are recorded in the dependency graph. For efficiency, we represent the dependency graph as a hash map from dependency query templates to dependent templates and their full dependency list. Therefore, determining newly satisfied FDQs can be performed quickly with simple lookup operations.

Algorithm 3 find_new_fdqs

Input: a query template Qt_i
Output: a set of newly discovered FDQs

```

queries_to_check = get_related_queries( $Qt_i$ )
queries_to_check = queries_to_check  $\cup$  { $Qt_i$ }
new_fdqs = {}
for all  $Qt_j \in$  queries_to_check do
  if !already_seen_deps( $Qt_j$ ) then
    p_mappings = get_prior_query_mappings( $Qt_j$ )
    if have_enough_mappings( $Qt_j$ ) then
      fdq = construct_fdq( $Qt_j$ , p_mappings)
      unresolved_deps = get_dependencies(fdq)
      add_to_dep_graph(unresolved_deps, fdq)
      mark_seen_deps(fdq)
      new_fdqs = new_fdqs  $\cup$  {fdq}
    end if
  end if
end for
return new_fdqs

```

Apollo ensures that there exists only one instance of an FDQ hierarchy throughout the system so that mapping updates affect both the FDQ and any FDQ structures that contain it. To do so, we track the FDQs that the system has constructed before, returning a previously constructed FDQ if applicable. During FDQ construction, dependency loops are detected and returned as dependency queries in an FDQ hierarchy. If all children of an FDQ are tagged as ADQs, or if an FDQ has no parameters and no children, then it is tagged as an ADQ and stored for use during cache reload (Section 3.4.2). Dependency queries are marked as unresolved dependencies on the FDQ and used to determine when an FDQ is ready for execution. Algorithm 4 shows how dependencies for known FDQs are tracked and used for predictive execution. After the execution of a given query template Qt_i , each dependent FDQ marks that dependency as satisfied. If all of an FDQ’s dependencies are now satisfied, we add it to a list of “ready FDQs”, resetting its dependencies so that they must be satisfied again before we determine the FDQ as being ready for future execution.

Algorithm 4 is used as part of a breadth-first approach to determine all runnable FDQs given the current query state. Apollo determines which FDQs are executable given the current system state and a newly executed query, adding them to the list of ready FDQs. Apollo then determines which other FDQs are executable given this FDQ list, repeating the process as necessary. This final list of FDQs is then executed in order, feeding result sets as parameters to dependent FDQs.

Algorithm 4 mark_ready_dependency

Input: an executed query Qt_i whose result set is now available

Output: a set ready_fdqs of FDQs ready for execution

```
ready_fdqs = {}
dependency_lists = get_dep_query_dlists( $Qt_i$ )
for all d_list  $\in$  dependency_lists do
  mark_dependency_satisfied(d_list,  $Qt_i$ )
  if all_deps_satisfied(d_list) then
    ready_fdqs = ready_fdqs  $\cup$  get_fdq(d_list)
    reset_dependencies(d_list)
  end if
end for
return ready_fdqs
```

3.2 Client Sessions

Apollo uses a client session consistency scheme [15], enabling its predictive cache to share cached results among clients and scale in the presence of write queries. In brief, each client has an independent session that guarantees that it accesses data at least as fresh as data it last read or wrote and that it efficiently shares cached entries with other clients.

Each client maintains a version vector (v_1, v_2, \dots, v_n) indicating its most recently accessed version v_i for each table R_i . Query results are stored in the cache and timestamped with a version vector (c_1, c_2, \dots, c_n) matching the version vector of the client that wrote it. When a client wants to execute a read query on a set of tables (R_1, R_2, \dots, R_n) , it checks if there exists an entry in the cache for that query with a version vector with $(c_1 \geq v_1, c_2 \geq v_2, \dots, c_n \geq v_n)$. If so, the client will retrieve and return the cached result, updating its client state for each of the tables to match that of the cached entry. If there is no such entry, the client will execute the query against the database, updating its version vector for each of the affected tables to match their versions in the database and storing the result in the cache. Write queries are never predictively executed (to prevent unnecessary rollbacks) and always execute against the database. After a client executes a write query, its version vector is updated to match the state of the database.

Since cache misses and write queries update a client's version vector, old cache entries may be stale under the client's new version vector. Therefore, if it is important to update a client's version vector only when strictly necessary, and by the minimum amount. As such, when a client could read two different versions of a cached key, Apollo will return the value for the cached key with a version vector that minimizes the distance from the client's version vector. Apollo uses a variety of optimizations to reduce the impact of write queries on predictive caching and system performance, discussed in Section 3.4.

Since a client's session is independent of the sessions of other clients, Apollo can easily scale horizontally. An individual client must route all of its requests to the same Apollo instance to maintain its session, but other clients and processes do not affect its session guarantees. Thus, extract, transform, load (ETL) processes, database triggers, and client write requests do not result in mass invalidations of cached data. Furthermore, Apollo instances do not need to communicate with each other to maintain sessions because a client's session is tracked by a single Apollo instance.

3.3 Publish–Subscribe Model

Since Apollo handles many concurrent clients, multiple clients may simultaneously try to execute the same read query. In these cases, it is beneficial to execute the query only once and return its result set to the waiting clients. Optimizing these queries is particularly important for predictive execution since a predicted query may not have finished execution before a client requests its result set.

Before executing a read query, Apollo consults a hash map to determine if a copy of the query is already executing. If so, Apollo blocks the query until the other query returns, passing along its result set. Otherwise, it will record an entry in the hash map with a semaphore for other clients and predictive pipelines to wait on. In this way, only one copy of a read query is executing at any time, including shared predictive query pipelines for multiple clients.

When Apollo determines that a client's query has multiple usable versions of its results cached, Apollo will use the earliest version regardless of whether another usable version is already being retrieved for a different client. Experimentally, we determined that it is better to retrieve results for earlier versions since reading later versions will result in large version vector updates for the client and may therefore cause misses for other cached results. Similarly, if Apollo must retrieve the result set from the database, Apollo will subscribe to any ongoing database retrievals of the same query.

3.4 Session-Aware Caching

Since write queries increment client version vectors, they preclude the client from reading any previously cached values. Therefore, if a client executes a write query after a predictive query is issued on that client's behalf, the predicted query results may be stale and unusable. If so, the system will have performed unnecessary work to execute and cache the query. To minimize the effects of writes on system performance, we avoid predictively executing queries whose results are likely to become stale before client queries can use their results (Section 3.4.1). Since ADQs can be executed at any time, we strive to keep valuable ADQs in the cache by reloading them if their results become outdated (Section 3.4.2).

3.4.1 Preventing Unusable Predictions. Apollo determines the likelihood of a write query or cache miss occurring using the query transition graph. Recall from Section 2.2 that each client has a single transition graph. However, by maintaining multiple independent transition graphs with different Δt intervals, we are able to determine the likelihood of a given query being executed by the client in each of these windows. Using this technique, we predict if a client will retrieve the results for a predictively executable query before its results become stale. Apollo will predictively execute and cache only query results that it deems are likely to be used.

To determine if predictively executing and caching a query's results will be helpful, Apollo first estimates the time it will take for the query to be executed and cached. Since all predictable queries are by definition FDQs, we use a simple estimate: the time to predictively execute an FDQ is given by the time it will take to execute its dependencies and the time to execute the FDQ itself. We calculate this estimate recursively: for a target FDQ, we return the maximum time to execute its dependency queries and add the time needed to execute the FDQ. In essence, this process returns the longest expected path from the child weighted by mean query runtimes. To provide an approximation of individual query runtimes, we use the mean execution time for each query

template. Although more sophisticated methods can be used [4, 45] to estimate query runtimes, we found that this method yields enough accuracy to determine the runtime of predicted query while still being performant.

Once the runtime for a given FDQ f has been determined (say t), Apollo looks up the client’s transition graph with smallest interval Δt where $\Delta t > t$. It then uses this graph to determine the likelihood of the client executing a query that would cause f ’s results — or the results of its dependencies — to become stale while f is executing. If this likelihood is sufficiently high (given the τ threshold), we avoid executing f to save on database execution costs. Therefore, only queries that are likely to be executed and useful to clients are predictively cached.

Although increasing the number of transition graphs per client necessitates additional processing of the query stream, we find that the simplicity of the query transition graph construction algorithm (Algorithm 1) combined with a configurable (but small) number of models per client results in low computational overhead for the system. Furthermore, since workloads [1, 42] tend to have a small number of unique query templates, the storage overhead is minimal.

3.4.2 Informed ADQ Reload. Write queries update a client’s version vector, and therefore provide an opportunity for optimization through informed query result reload. As ADQ dependencies are always satisfied and can be executed at any time, we immediately reload valuable ADQ hierarchies after a client executes a write query. Since there can be many ADQs and reloading a hierarchy may be expensive for the database to execute, we limit ADQ reload to only those predictions for query templates considered valuable according to the cost function $cost(Q_t) = P(Q_t) \cdot mean_rt(Q_t)$.² Specifically, the estimated $cost$ of an ADQ on the system is given by the probability of the ADQ executing and the estimated ADQ runtime. If the cost of the ADQ exceeds a predefined threshold α , we reload it into the cache. We discuss α and its effects further in Section 4.7.

4 PERFORMANCE EVALUATION

In this section, we present the system setup used to conduct experiments followed by performance results. Apollo is compared against Memcached [19], a popular mid-tier cache used in database and storage systems, as well as the Fido predictive cache [34]. We compare these systems using average query response time and tail latencies, which have been observed to contribute significantly to user experience and indicate concurrent interaction responsiveness [28].

The Fido engine serves as a drop-in replacement for Apollo’s prediction engine, and uses Palmer et al.’s associative-memory technique [34] for query prediction, scanning client query streams to predict upcoming queries. Fido-like approaches have been employed to prefetch objects in databases [8]. Fido’s implementation-independent middleware prediction engine makes it particularly well-suited as a comparison point against Apollo.

The remainder of this section is organized as follows. Section 4.1 describes our experiments’ setup and Section 4.2 provides performance experiments for TPC-W. In Section 4.3, we use TPC-C to assess Apollo’s scalability under increasing client load. Section 4.4 showcases Apollo’s ability to adapt to changing workloads using online learning. Geographic latency experiments and multi-Apollo instance experiments are shown in Sections 4.5

²Note that the techniques in Section 3.3 apply; shared query dependencies and overlapping client query submissions will not result in multiple executions of ADQs.

and 4.6 respectively, and Section 4.7 presents a sensitivity analysis of Apollo’s configurable parameters.

4.1 Experimental Setup

Our experiments use a geo-distributed setup in which Amazon EC2 nodes are located in the US-East (N. Virginia) region for: (i) Apollo with 16 virtual CPUs, 64 GB of RAM and a 50 GB SSD (ii) Memcached on a machine with 2 virtual CPUs, 4 GB of RAM, and a 50 GB SSD (iii) a node with concurrent clients running our benchmarks with 16 virtual CPUs, 64 GB of RAM, and a 50 GB SSD. We deploy a database machine in the US-West (Oregon) region for our experiments, which has 16 virtual CPUs, 64 GB of RAM, a 250 GB SSD, and uses MySQL v5.6 as the database. For each experiment, Memcached uses a cache size 5% of the size of the remote database to demonstrate that Apollo is effective with limited cache space. All results presented are the average over at least five independent runs, with bars around the means representing 95% confidence intervals.

Our experiments have three primary configurations: the Memcached configuration (in which the cache has been warmed for 20 minutes prior to benchmarking), the Apollo caching configuration, and the Fido prediction engine configuration [34]. In the Memcached configuration, we check for query results in the cache and forward queries on cache misses to the remote database, caching the retrieved query results. The Apollo and Fido configurations also load query results into the cache after they execute a read-only query on the remote database, but Apollo uses the predictive framework from Section 2 and Fido uses its own predictive engine, which is detailed below.

Unlike Apollo, Fido functions on an individual query level rather than on query templates. More concretely, if queries Q_1, Q_2, \dots, Q_n are present in a client’s query stream, Fido looks for a stored pattern that is prefixed by them, say $Q_1, Q_2, \dots, Q_n, P_1, P_2, \dots, P_m$, proceeding to predictively execute P_1, P_2, \dots, P_m and cache their results. In contrast to Apollo’s online learning capabilities, Fido requires *offline* training to make predictions. We provide Fido with client workload traces twice the length of the experiment interval to serve as its training set for comparison against a *cold start* Apollo. Additionally, we let Fido make up to 10 predictions for each matched prefix.

In all configurations, clients use session guarantees (Section 3.2) and queries executed at the remote database have their result sets immediately cached in Memcached. Thus, the difference in caching performance between the configurations is due to caching benefits provided by the query prediction engines.

Our experiments aim to answer three key questions. First, can Apollo analyze incoming queries and learn patterns within a workload? Second, are Apollo’s predictive caching capabilities effective in reducing query round-trip time by avoiding costly database query executions? Third, can Apollo’s predictive framework scale with an increasing number of clients? We present performance results in the next sections that include answers to these questions.

4.2 TPC-W Benchmark

The TPC-W Benchmark [42] generates a web commerce workload by having emulated browsers interact with servlets that serve webpages. The webpages require persistent data from storage so servlets execute database queries against the remote database to generate webpage content. The TPC-W benchmark includes 14 different web interactions for clients (e.g., Best Sellers, Order Inquiry) each with their own distinct set of queries. For a given

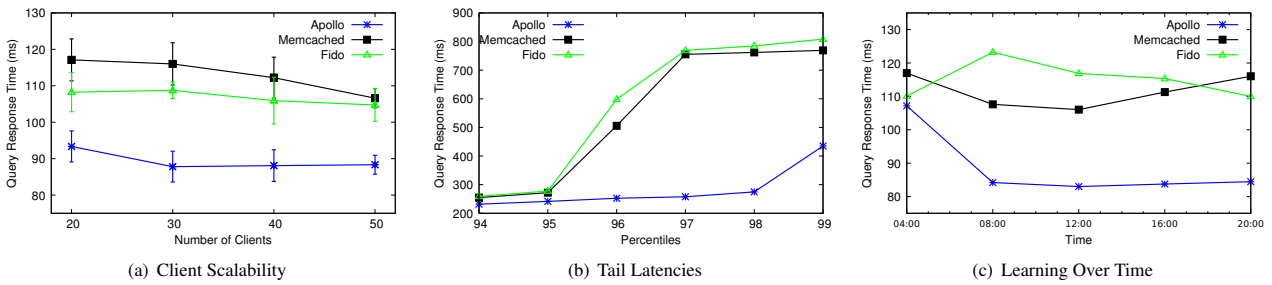


Figure 5: Experiment results for 20 minute TPC-W runs using Apollo, Fido, and Memcached (no prediction engine).

client, the next web interaction is chosen probabilistically based on the previous interaction. We use a popular implementation [35] of the TPC-W Benchmark specification.

The TPC-W benchmark represents an important use case for Apollo since even small changes in latency can significantly impact web traffic [20] and sales [30]. Further, it serves as a challenging workload for Apollo due to its inherent randomness and large number of different queries. This randomness serves to test the viability of Apollo’s predictive framework under a variable workload.

We generated a 33 GB TPC-W database with 1,000,000 items. We measured Apollo’s performance using the TPC-W benchmark browsing mix executed for 20 minute measurement intervals while scaling-up the number of clients using our default TPC-W parameters discussed in Section 4.7.

4.2.1 Performance Results. Figure 5(a) shows Apollo’s performance for an increasing number of clients compared to Memcached and Fido. Apollo significantly outperforms both Fido and Memcached, enjoying a large response time reduction of up to 33% over Memcached and 25% over Fido. Fido has slightly lower response time than Memcached due to query-instance level predictive caching but is unable to recognize query template patterns and generalize to unseen queries, precluding it from being competitive with Apollo. In the case of Memcached, we see its warmed cache offers little advantage over Apollo’s and Fido’s cold starts — invalidation and randomness limit the effects of cache warming.

Each configuration shows a reduction in response time as the number of clients increase, a consequence of the shared cache between clients. However, shared caching is unable to compete with our predictive caching scheme as in a shared cache, a client must incur a cache miss, execute, and then store query results before others can use it. Consequently, Apollo’s techniques of query prediction and informed ADQ reload prove superior, even as the client load is scaled up.

Figure 5(b) shows the distribution of tail response times for each of the experimental configurations for 50 client TPC-W runs. Apollo’s response times are significantly lower than any of the other methods, particularly for the higher percentiles, due to an improvement in cache hits. At the 97th percentile, Apollo reduces tail latencies by 1.8x over Memcached and Fido. Again, Fido tends to perform about as well as Memcached, despite its large training set size, as it cannot generalize its patterns to query templates for FDQ prediction and query reload.

Figure 5(c) shows average query response times in 4 minute intervals. We see that Apollo exhibits a downward trend in response time from the start of the measurement interval as it effectively

learns query correlations and parameter mappings, resulting in an improvement of 30% over its average response time during the first four minutes. Although the other systems’ performance oscillates according to workload patterns, they do not learn query patterns — their final average query response times are comparable to that incurred in their first few minutes.

To ensure that Apollo can provide these response time reductions without undue resource overhead, we added instrumentation to determine the time and memory needed to find and construct new FDQs. On average, it takes less than 1% of response time to discover new FDQs given a newly executed query, and less than 2% of response time to construct an FDQ. We have observed that Apollo uses scant system resources, requiring only 1.5% the amount of memory used by the database for tracking the transition graph and query parameter mappings. Apollo’s predictive techniques submit an additional 25% more queries to the remote database compared to the Memcached configuration. Apollo’s intelligent query caching techniques place little additional load on the remote database and use meager resources, while still providing substantially lower average query response times than both Fido and Memcached.

To answer the performance questions we had posed earlier in Section 4.1, Apollo is indeed able to make accurate and useful predictions for what to cache, predicting and retaining important result sets in the cache for longer without significant computation or memory overhead.

4.3 TPC-C Benchmark

The TPC-C Benchmark emulates an order-entry environment in which multiple clients execute a mix of transactions against a database system [1]. Each of these clients functions as a store-front terminal, which submits orders for customers, confirms payments, and updates stock levels. In contrast to TPC-W’s workload, TPC-C’s OLTP workload features many short-running queries which avoid contention by reduced locking of significant parts of the database. As such, the TPC-C benchmark serves to directly test the scalability of Apollo.

The TPC-C specification has two read-only transactions, Stock Level and Order Status, both of which present opportunities for predictive execution. Since the goal of our experimentation with TPC-C is to show the scalability of predictive execution under high numbers of clients, we scale up the mix of read-only transactions to 95% with updates making up the remaining 5%. In doing so, Apollo must track, construct, and execute far more opportunities for predictive queries than in the TPC-W experiments. Thus, this experiment’s purpose is to show how well Apollo can handle hundreds of clients executing predictive queries simultaneously.

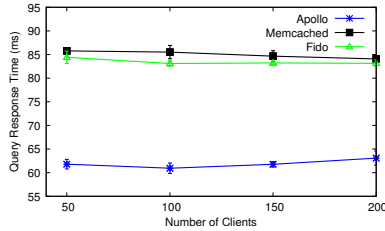


Figure 6: Experiment results for 20 minute TPC-C runs using Apollo, Fido, and Memcached (no prediction engine).

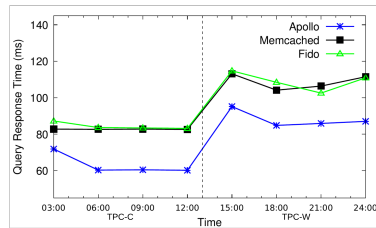


Figure 7: Experiment results for changing the workload from TPC-C to TPC-W using Apollo, Fido, and Memcached (no prediction engine).

In our experiments, we use the OLTPBench TPC-C implementation from Difallah et al [18]. To properly assess scalability, we modified the read/write mix, with a final percentage of 5% Payments, 47.5% Order Status, and 47.5% Stock Level Transactions. This mix forces the prediction engines to construct and execute significantly more predictive queries.

A TPC-C database of size 100 GB with 1000 warehouses was generated and loaded into a US-West MySQL instance using the data generation mechanism of OLTPBench. For the following experiments, we used our default TPC-C parameters (discussed in Section 4.7) with a 5% write mix. We choose the warehouse parameter in our queries according to a uniform distribution, which results in more predictive executions than a skewed Zipf distribution — recall that Apollo will not predictively execute queries that are already cached (Section 3.3).

4.3.1 Performance Results. Figure 6 shows the scalability of Apollo, Fido, and Memcached for increasing numbers of clients. Apollo exhibits a significantly lower average response time than Memcached and Fido, even as the number of clients, and therefore the number of predictive query executions, increases. Apollo’s efficient data structures and algorithms for tracking and prediction allow scaling even with a large number of clients. Fido and Memcached perform about the same, even as we increase the number of clients. With a large database, query parameters are highly variable and rarely repeated, causing Fido’s non-template approach to see few queries from its training set and in turn reducing prediction accuracy. As the number of clients increase, the positive effects of shared caching dwarf that of Fido’s predictions, resulting in similar performance characteristics between Fido and Memcached.

These results show that Apollo can deliver significant performance gains while scaling effectively to hundreds of concurrent clients continuously executing predictive queries.

4.4 Adapting to Changing Workloads

To assess Apollo’s ability to adapt to changing workloads, we conducted an experiment in which the workload was changed from our TPC-C workload described in Section 4.3 to TPC-W partway through the experiment (Figure 7). We see that Apollo quickly learns predictions for the TPC-C workload, resulting in the performance gains shown in Figure 6. By contrast, Fido and Memcached have relatively constant performance during the TPC-C run since they are unable to generalize and make effective predictions for upcoming queries (Section 4.3).

Once the workload switches, shown by a dashed vertical line in Figure 7, each configuration experiences a brief penalty in performance because the predictive engines cannot make any predictions for queries in the new workload, and no TPC-W queries are cached. However, Apollo quickly returns to its typical performance on TPC-W (Figure 5(a)) since it uses online learning to discover query patterns. Fido and Memcached perform similarly after the switch since Fido is unable to make predictions for an untrained workload. Although Fido was trained for TPC-C in this experiment, we note that its performance is comparable to an appropriately trained Fido on the TPC-W portion. This observation further highlights the ineffectiveness of Fido’s prediction scheme for the correlated query patterns, which Apollo excels at predicting.

4.5 Geographic Latency Testing

To assess the effects of different geographic latencies between Apollo and the database, we deployed TPC-W databases in the US-East and Canada regions. Because Apollo, the cache and the benchmark machine are all located in the US-East region, the first configuration tests a “local” deployment, in which latency among the machines is minimal (a few milliseconds). The second configuration tests moderate latencies of 20 ms.

In both configurations (Figures 8(a) and 8(b)), we see that Apollo preserves its lead over the other systems despite limited geographic latency. Apollo reduces query response time by up to 50% in the US East region and by up to 40% in the Canada region. This improvement in the performance gap compared to the higher latency experiments is because cache misses in low latency environments have a larger effect on average performance than when latency is high. The reason for this effect is that Apollo’s advantage when caching expensive queries becomes even more significant with reduced latency; prioritizing expensive and frequently executed queries results in a substantial improvement and failure to predictively cache them (as in Memcached’s and Fido’s case) results in a relatively larger performance degradation.

These results are not to be interpreted as Apollo is “best” in a local setting with near zero latency — the *total* response time savings for the remote settings are larger than that of the local setting. Apollo provides substantial reductions in average and total response time in both settings, resulting in an enhanced user experience.

4.6 Multiple Apollo Instances

Apollo can scale to high loads by partitioning clients among multiple Apollo engine instances and cache stores. Each Apollo engine maintains a consistent session for each client connected to it, without interacting with the other instances or a centralized session manager.

To determine Apollo’s scaling characteristics, we deployed Apollo on less powerful m4.xlarge EC2 instances with 4 vCPUs

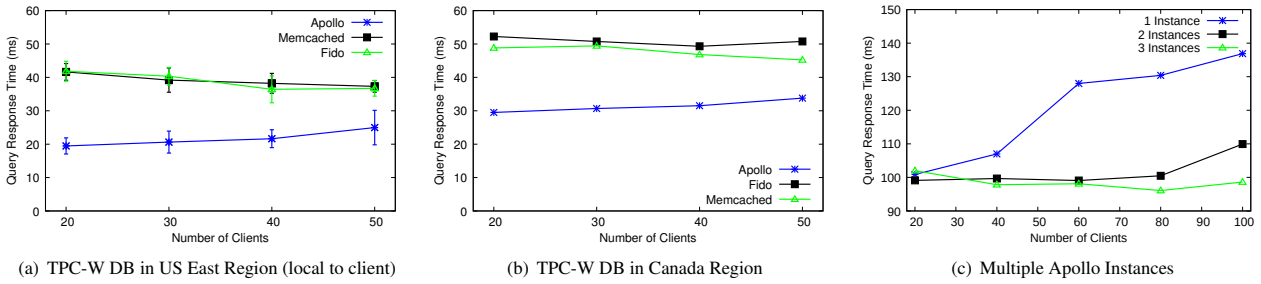


Figure 8: Experiment results for 20 minute TPC-W runs in different geographic regions and when using multiple Apollo instances.

and 16 GB of RAM. We use these less powerful machines as they are individually unable to handle large numbers of clients, necessitating a horizontal scale-out for Apollo instances. We test three different Apollo configurations: one with a single Apollo instance, another with two Apollo instances, and a third with three Apollo instances. Each Apollo instance is given its own dedicated cache, thereby avoiding the need to synchronize version vectors across instances to maintain client sessions. Clients are evenly distributed and pinned to Apollo instances.

The results of the experiment are shown in Figure 8(c). As the client load increases, we see that it quickly overwhelms the 1-instance Apollo configuration, resulting in a large increase in query response time. The 2-instance and 3-instance Apollo configurations show significantly improved scalability, though eventually the 2-instance configuration begins to show a similar upward trend in response time.

We observed that the 2-instance query response time at 20 clients is slightly lower than that of the 3-instance configuration. This effect is primarily due to splitting the clients across three machines rather than two, resulting in the 3-instance configuration receiving fewer queries to learn from. ADQs are shared among clients, which results in longer learning times to reach a steady state with fewer clients. With a larger number of clients, the increase in the amount of data to learn from and the computing power available result in improved performance over the 2-instance configuration.

Although having multiple Apollo instances share models and training data would reduce the effects of training data splitting, the trade-off is increased synchronization between otherwise independent nodes. We eschew this approach for two reasons. First, clients should be split across Apollo instances only when a single instance cannot handle the load. As seen in Figure 8(c), Apollo receives enough training data from clients well before it reaches its load capabilities, even on a less powerful machine. Second, slightly increased training times are a small price to pay for horizontal scalability. Addressing scalability issues in production systems is challenging — learning for a few more minutes is a simple and inexpensive solution to the insufficient data problem.

These multi-instance experiments demonstrate Apollo’s ability to scale to large numbers of clients through horizontal scaling and client-session consistency semantics. Since separate Apollo instances do not need to communicate, Apollo demonstrates excellent scaling characteristics.

4.7 Sensitivity Analysis

A key feature of Apollo is its ability to be configured to operate under different workloads and system deployments. This configurability is enabled by the provision of parameters that can be set according to a particular workload and deployment. In this section, we discuss the effects of these parameters on the overall performance of the system as well as our choice of their default settings.

For TPC-W, our default parameter choices were: $\Delta t = 15s$, $\tau = 0.01$, and a reload cost threshold of $\alpha = 0$. Per the specification [42], TPC-W uses randomized think times with a mean of 7s. Each client has its own application state, which is determined through a probabilistic transition matrix. Therefore, a client’s web interactions do not generate a pre-determined chain of queries.

The maximal time separation Δt and minimum probability threshold τ are correlated. As Δt decreases, the probability of correlated query templates executing within this interval also decreases, thereby requiring a lower τ value to capture the relationship between query templates. Similarly, as Δt increases, the probability of two correlated query templates executing within this interval also increases, so higher τ values are sufficient. If Δt is large and τ is small then it is likely that spurious relationships between query templates will be discovered, but such spurious relationships are filtered out by Apollo’s parameter mapping verification period, and are therefore seldom predictively executed. Since TPC-W’s workload bottlenecks on the database, Apollo filters out the spurious correlations in exchange for discovering as many relationships as possible. To do so, we set a high value of $\Delta t = 15s$ and a low threshold of $\tau = 0.01$. These values were empirically confirmed to yield the best results.

In Section 3.4.2, we defined α to be the minimum cost that an ADQ must have to be reloaded. Note that the cost of an ADQ is the mean response time multiplied by the probability of the query executing. Hence as α is increased, only ADQs that are both popular and expensive are reloaded. We experimented with different values of α and found that for small values of α (less than 5% of the mean query response time), there was little change in query response time. However, as α continued to be increased past this threshold, the mean query response time grew by over 10%, as valuable ADQs were not reloaded. To ensure that all ADQs were reloaded, α was set to 0 in our experiments.

We observed similar trends with Δt and τ in TPC-C as in TPC-W; therefore, our default parameter choices for TPC-C were the same. We left Δt large and τ small to place additional pressure on Apollo’s parameter mapping filtering functionality. These values were empirically confirmed to yield the best results.

In our experiments, we used a cache 5% the size of the database. We observed that increasing the cache beyond this size did not affect the relative performance differences between Apollo, Memcached, and Fido.

5 RELATED WORK

Fido [34], detailed in Section 4.1, uses an associative memory model for predictive prefetching in a traditional client/server database system. Query patterns in Apollo are tracked at the query template level so a single relationship in Apollo can map to many in Fido. Tracking individual data object accesses, or parameterized queries, means that if Fido has not previously seen a particular parameterized query it will not be able to make a prediction. In contrast, if Apollo has seen the query template (regardless of parameters), it can infer correlation between queries and predictively execute. As Fido requires offline training, it cannot adapt to changes in object access patterns. As we have shown, the online nature of Apollo's Markov model allows it to dynamically change over time and thus adapt to new query patterns.

Keller et al. [24] describe a distributed caching system for databases, which uses approximate cache descriptions to distribute update notifications to relevant sites and execute queries locally on caches. Each site's cached data is tracked using query predicates. Apollo differs from this work in that we focus on the predictive execution of consequent queries derived from query patterns, which Keller et al. do not consider.

Scalpel [10] tracks queries at a database cursor level, intercepting open, fetch and close cursor operations within the JDBC protocol. Since the JDBC API is translated to database specific protocols, Scalpel functions as a client-side cache rather than a mid-tier shared cache like Apollo. Unlike Apollo's online learning model, Scalpel requires offline training to find the patterns that it uses for query rewriting and prefetching. Scalpel employs aggressive cache invalidation on writes and at the start of new transactions, which differs from Apollo's client-centric consistency model. Given that Apollo supports mid-tier shared caching across multiple clients, this makes Scalpel unsuitable for comparison against Apollo.

Pavlo et al. [7] implement Markov models in H-Store and use them to optimize transaction execution for distributed database physical design. The system constructs a series of Markov models for stored procedures and monitors the execution paths under a set of input parameters. Their model can be leveraged to determine a base partition for stored procedures and to lock only partitions that are predicted to be accessed during procedure execution. Apollo operates beyond this stored procedure context, and provides benefits through caching future queries rather than by analyzing query execution paths.

DBProxy [5] is a caching system developed by IBM to cache query results on edge nodes. DBProxy uses multi-layered indexes and query containment to match queries to results, evicting stale and unused results. Its single session guarantees differ from Apollo's per-client sessions and limit scalability, in addition to not using online learning or predictive caching to improve performance.

Ramachandra et al. [36] propose a method for semantic prefetching by analyzing the control flow and call graph of program binary files. Given the source code for a database application, the system analyzes and modifies it, adding prefetch requests into the code as soon as the parameters are known and query execution guaranteed. Since this work is limited to requiring access to the source code of

application binaries, it only works for fixed workloads. Because Apollo analyzes query streams, it is able to adapt to changing query patterns over time.

Although proprietary middleware caching solutions have been developed [9, 16, 26], they do not use predictive analytics to identify future queries and preload them in the cache.

Scheuermann et al. [40] propose the Watchman cache management system, which uses query characteristics to improve cache admission and replacement algorithms. Unlike Apollo, Watchman does not discover query patterns for use in predictive execution and instead focuses solely on cache management.

Holze et al. [23] have broached the idea of modeling workloads using Markov models, but such work focuses only on determining when an application's workload has been altered rather than relying on statistical models for caching purposes, like Apollo. They suggest a Markov model as a means to achieve an automatic database, enabling features such as self-configuration, self-optimization, self-healing, and self-protection. In contrast, Apollo uses Markov models of user workloads to predict future queries and enables predictive query caching.

Promise [37] is a theoretical framework for predicting query behaviour within an OLAP database. Promise uses Markov models to predict user query behaviour by developing state machines for parameter value changes and transitions between OLAP queries, but does not consider direct parameter mappings, FDQ hierarchies, or pipelining predictions. Unlike Apollo, Promise does not validate its techniques through a system implementation.

Recent research in approximate query processing [6, 44] has proposed using previous query results as a means for approximating the answer to future queries. These works develop statistical methods to provide accurate, approximate answers and error bounds for upcoming queries, which differs from Apollo's focus on learning parameter mappings for predictive caching.

In the view selection problem [12], one must decide on a set of views to materialize so that execution of the workload minimizes some cost function and uses fixed amount of space. Most work in this area requires knowledge of the workload ahead of time [2, 22], with the remainder not considering machine learning techniques for uncovering patterns for use in view selection [17, 25].

XML XPath templates have some similarities to query templates [31], but are not used for online learning in predictive execution and caching. Instead, XPath views are selected using offline training in a warm-up period [29, 43], similar to that of Fido [34]. Similar ideas have been explored to cache dynamic HTML fragments [14].

6 CONCLUSION

In this paper, we propose a novel method to determine and leverage hidden relationships within a database workload via a predictive learning model. We present the Apollo system, which exploits query patterns to predictively execute and cache query results. Apollo's online learning method makes it suitable for different workloads and deployments. Experimental evaluation demonstrates that Apollo is a scalable solution that efficiently uses a cache and outperforms both Memcached, an industrial caching solution for different workloads and the popular Fido predictive cache.

ACKNOWLEDGMENTS

Funding for this project was provided by the Cheriton Graduate Scholarship, Ontario Graduate Scholarship, and the Natural

Sciences and Engineering Research Council of Canada. We are grateful for compute resource support from the AWS Cloud Credits for Research program.

REFERENCES

- [1] February 2010. The Transaction Processing Council. TPC-C Benchmark (Revision 5.11). <http://www.tpc.org/tpcc/>. (February 2010).
- [2] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. 2000. Automated Selection of Materialized Views and Indexes in SQL Databases. In *PVLDB (VLDB '00)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 496–505. <http://dl.acm.org/citation.cfm?id=645926.671701>
- [3] Akamai. 2010. New Study Reveals the Impact of Travel Site Performance on Consumers. <https://www.akamai.com/us/en/about/news/press/2010-press/new-study-reveals-the-impact-of-travel-site-performance-on-consumers.jsp>. (2010).
- [4] M. Akdere, U. Çetintemel, M. Riondato, E. Ufpa, and S. B. Zdonik. 2012. Learning-based Query Performance Modeling and Prediction. In *2012 IEEE 28th International Conference on Data Engineering*. 390–401. <https://doi.org/10.1109/ICDE.2012.64>
- [5] K. Amiri, Sanghyun Park, R. Tewari, and S. Padmanabhan. 2003. DBProxy: a dynamic data cache for web applications. In *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)*. 821–831. <https://doi.org/10.1109/ICDE.2003.1260881>
- [6] Christos Anagnostopoulos and Peter Triantafyllou. 2017. Efficient scalable accurate regression queries in IN-DBMS analytics. *Proceedings - International Conference on Data Engineering (2017)*, 559–570. <https://doi.org/10.1109/ICDE.2017.111>
- [7] Stanley Zdonik Andrew Pavlo, Evan P.C. Jones. 2012. On Predictive Modeling for Optimizing Transaction Execution in Parallel OLTP Systems. *PVLDB* 5, 2 (2012), 85–96.
- [8] P.A. Bernstein, S. Pal, and D.R. Shutt. 2009. Prefetching and caching persistent objects. (June 30 2009). <https://www.google.com/patents/US7555488> US Patent 7,555,488.
- [9] Christof Bornhövd, Mehmet Altinel, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh, and Berthold Reinwald. 2003. DBCache: Middle-tier Database Caching for Highly Scalable e-Business Architectures. In *SIGMOD (SIGMOD '03)*. ACM, New York, NY, USA, 662–662. <https://doi.org/10.1145/872757.872849>
- [10] Ivan Bowman and Kenneth Salem. 2004. Optimization of query streams using semantic prefetching. *SIGMOD* (2004), 179–190.
- [11] Nathan Bronson, Zach Amsden, George Cabrera, et al. 2013. TAO: Facebook’s distributed data store for the social graph. *Usenix ATC* (2013), 49–60. <https://doi.org/10.1145/2213836.2213957>
- [12] Rada Chirkova, Alon Y Halevy, and Dan Suciu. 2001. A formal perspective on the view selection problem. In *VLDB*, Vol. 1. 59–68.
- [13] James C Corbett, Jeffrey Dean, Michael Epstein, and Andrew Fikes. 2012. Spanner : Google’s Globally-Distributed Database. *OSDI* (2012), 1–14. <https://doi.org/10.1145/2491245>
- [14] Anindya Datta, Kaushik Dutta, Helen Thomas, Debra V, Krithi Ramamritham, and Dan Fishman. 2001. A comparative study of alternative middle tier caching solutions to support dynamic web content acceleration. In *In International Conference on Very Large Data Bases (VLDB)*. 25.
- [15] Khuzaima Daudjee and Kenneth Salem. 2004. Lazy Database Replication with Ordering Guarantees. In *ICDE*. 424–435.
- [16] Louis Degenaro, Arun Iyengar, Ilya Lipkind, and Isabelle Rouvellou. 2000. A Middleware System Which Intelligently Caches Query Results. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware '00)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 24–44. <http://dl.acm.org/citation.cfm?id=338283.338285>
- [17] Prasad M Deshpande, Karthikeyan Ramasamy, Amit Shukla, and Jeffrey F Naughton. 1998. Caching multidimensional queries using chunks. In *ACM SIGMOD Record*, Vol. 27. ACM, 259–270.
- [18] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. 2013. OLTP-Bench: An extensible testbed for benchmarking relational databases. *PVLDB* 7, 4 (2013), 277–288.
- [19] Brad Fitzpatrick. 2016. MemCached. (4 2016). <https://memcached.org/Memcached Caching Software>.
- [20] Brady Forest. 2009. Bing and Google Agree - Slow Pages Lose Users. <http://radar.oreilly.com/2009/06/bing-and-google-agree-slow-pag.html>. (2009).
- [21] Google. 2017. Google’s Edge Network. <https://peering.google.com/infrastructure>. (2017).
- [22] Himanshu Gupta. 1997. Selection of Views to Materialize in a Data Warehouse. In *ICDT (ICDT '97)*. Springer-Verlag, London, UK, UK, 98–112. <http://dl.acm.org/citation.cfm?id=645502.656089>
- [23] Marc Holze and Norbert Ritter. 2007. Towards Workload Shift Detection and Prediction for Autonomic Databases. In *Proceedings of the ACM First Ph.D. Workshop in CIKM (PIKM '07)*. ACM, New York, NY, USA, 109–116. <https://doi.org/10.1145/1316874.1316892>
- [24] Arthur M. Keller and Julie Basu. 1996. A Predicate-based Caching Scheme for Client-server Database Architectures. *VLDBJ* 5, 1 (Jan. 1996), 035–047. <https://doi.org/10.1007/s007780050014>
- [25] Yannis Kotidis and Nick Roussopoulos. 1999. DynaMat: a dynamic view management system for data warehouses. In *ACM SIGMOD Record*, Vol. 28. ACM, 371–382.
- [26] Per-Ake Larson, Jonathan Goldstein, and Jingren Zhou. 2004. MTCache: Transparent Mid-Tier Database Caching in SQL Server. In *ICDE (ICDE '04)*. IEEE Computer Society, Washington, DC, USA, 177–.
- [27] Ang Li, Xiaowei Yang, Srikanth Kandula, and Ming Zhang. 2010. CloudCmp: Comparing Public Cloud Providers. *Proceedings of the 10th annual conference on Internet measurement - IMC '10* (2010), 1. <https://doi.org/10.1145/1879141.1879143>
- [28] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. 2014. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '14)*. ACM, New York, NY, USA, Article 9, 14 pages. <https://doi.org/10.1145/2670979.2670988>
- [29] Kostas Lillias and Evaggelia Pitoura. 2008. Cooperative XPath Caching. In *SIGMOD (SIGMOD '08)*. ACM, New York, NY, USA, 327–338. <https://doi.org/10.1145/1376616.1376652>
- [30] Greg Linden. 2006. Make Data Useful. <http://www.gdunchamp.com/media/StanfordDataMining.2006-11-28.pdf>. (2006).
- [31] Bhushan Mandhani and Dan Suciu. 2005. Query Caching and View Selection for XML Databases. In *PVLDB (VLDB '05)*. VLDB Endowment, 469–480. <http://dl.acm.org/citation.cfm?id=1083592.1083648>
- [32] Oracle. 2017. Java SE 8 JDBC API. <https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/>. (2017).
- [33] Oracle. 2017. MySQL. <https://www.mysql.com/>. (2017).
- [34] Mark Palmer and Stanley B. Zdonik. 1991. Fido: A Cache That Learns to Fetch. In *VLDB (VLDB '91)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 255–264.
- [35] Jose Pereira. 2016. TPC-W Implementation. (4 2016). University of Minho’s implementation of TPC-W.
- [36] Karthik Ramachandra and S. Sudarshan. 2012. Holistic Optimization by Prefetching Query Results. In *SIGMOD (SIGMOD '12)*. ACM, New York, NY, USA, 133–144. <https://doi.org/10.1145/2213836.2213852>
- [37] Carsten Sapia. 2000. PROMISE: Predicting query behavior to enable predictive caching strategies for OLAP systems. In *International Conference on Data Warehousing and Knowledge Discovery*. Springer, 224–233.
- [38] Mahadev Satyanarayanan, Paramvir Bahl, Ramon Caceres, and Nigel Davies. 2009. The Case for VM-Base Cloudlets in Mobile Computing. *Pervasive Computing* 8, 4 (2009), 14–23. <https://doi.org/10.1109/MPRV.2009.82>
- [39] Mahadev Satyanarayanan, Zhuo Chen, Kiryong Ha, Wenlu Hu, Wolfgang Richter, and Padmanabhan Pillai. 2014. Cloudlets: at the Leading Edge of Mobile-Cloud Convergence. *Proceedings of the 6th International Conference on Mobile Computing, Applications and Services* (2014), 1–9. <https://doi.org/10.4108/icst.mobica.2014.257757>
- [40] Peter Scheuermann, Junho Shim, and Radek Vingralek. 1996. WATCHMAN: A Data Warehouse Intelligent Cache Manager. In *Proceedings of the 22th International Conference on Very Large Data Bases (VLDB '96)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 51–62. <http://dl.acm.org/citation.cfm?id=645922.758367>
- [41] Jeff Shute, Mircea Oancea, Stephan Ellner, et al. 2012. F1: the fault-tolerant distributed RDBMS supporting Google’s ad business. In *SIGMOD*. 777–778.
- [42] TPC. 2000. TPC Benchmark W (Web Commerce). <http://www.tpc.org/tpcw>. (2000).
- [43] Liang Huai Yang, Mong Li Lee, and Wynne Hsu. 2003. Efficient Mining of XML Query Patterns for Caching. In *PVLDB (VLDB '03)*. VLDB Endowment, 69–80. <http://dl.acm.org/citation.cfm?id=1315451.1315459>
- [44] Barzan Mozafari Yongjoo Park, Ahmad Shahab Tajik, Michael Cafarella. 2017. Database Learning: Toward a Database that Becomes Smarter Every Time. *SIGMOD* (2017), 587–602.
- [45] E. E. Yusufoglu, M. Ayyildiz, and E. Gul. 2014. Neural network-based approaches for predicting query response times. In *2014 International Conference on Data Science and Advanced Analytics (DSAA)*. 491–497. <https://doi.org/10.1109/DSAA.2014.7058117>

Interactive Rule Refinement for Fraud Detection

Tova Milo
Tel Aviv University
milo@post.tau.ac.il

Slava Novgorodov
Tel Aviv University
slavanov@post.tau.ac.il

Wang-Chiew Tan
Recruit Institute of Technology
wangchiew@recruit.ai

ABSTRACT

Credit card frauds are unauthorized transactions that are made or attempted by a person or an organization that is not authorized by the card holders. Fraud with general-purpose cards (credit, debit cards etc.) is a billion dollar industry and companies are therefore investing significant efforts in identifying and preventing them.

It is typical to deploy mining and machine learning-based techniques to derive rules. However, such rules may not always capture the semantic reasons underlying the frauds that occur. For this reason, credit card companies often employ domain experts to manually specify rules that exploit general or domain knowledge for improving the detection process. Over time, however, as new (fraudulent and legitimate) transactions arrive, these rules need to be updated and refined to capture the evolving (fraud and legitimate) activity patterns. The goal of the RUDOLF system described in this paper is to guide and assist domain experts in this challenging task. RUDOLF automatically determines the “best” adaptation to existing rules to capture all fraudulent transactions and, respectively, omit all legitimate transactions. The proposed modifications can then be further refined by users and the process can be repeated until they are satisfied with the resulting rules. We show that the problem of identifying the best candidate adaptation is NP-hard in general and present PTIME heuristic algorithms for determining the set of rules to adapt. We have implemented our algorithms in RUDOLF and show, through experiments on real-life datasets, the effectiveness and efficiency of our solution.

1 INTRODUCTION

A *credit card fraud* is an unauthorized transaction made or attempted by an individual or organization who is not authorized by the card holder to use a credit card to perform the electronic payment. Fraud with general-purpose cards (credit, debit cards etc.) is a billion dollar industry. In fact, several independent news articles and studies that were carried out (e.g., [1, 2]) corroborate that there is a consistent, fast-growing, and upward trend on the total global payment-card frauds. Detecting and deterring credit card frauds are therefore of extreme importance to credit card companies. A core part of operations behind every credit card company is to (automatically) detect fraudulent transactions among the new transactions (e.g., [3]) that are received everyday.

To this end, models based on data mining and machine-learning techniques (e.g. [4–6]) have been used. A typical approach is to score each transaction where transactions whose scores are above a threshold are classified as fraudulent. However, the models and scoring system do not always have high precision and recall. Fraudulent transactions may be missed by the models and, likewise, legitimate transactions may be wrongly identified as fraudulent. The derived threshold also do not provide a semantic explanation of the underlying causes of the frauds. It is for this reason that credit card companies typically rely on *rules* that are

carefully specified by domain experts in addition to models for automatically determining fraudulent transactions.

Intuitively, a rule describes a set of transactions in the database and the goal is to arrive at a set of rules that, together with the automatically derived scores, captures precisely the fraudulent transactions. The use of rules written by users has the advantage that it allows employing general or domain knowledge to handle rare special cases.

Writing rules to capture precisely fraudulent transactions is a challenging task that is exacerbated over time as the types of fraudulent transactions evolve or as new knowledge is learnt. Typically, a set of rules that were curated by users already exists and the rules work well for capturing fraudulent transactions up to a certain day. However, these rules need to be adapted over time to capture new types of frauds that may occur. For example, there may be new reported fraudulent transactions coming from a certain type of store at a certain time that did not occur before and hence, not caught by existing rules. Analogously, there may be some transactions that were identified by the existing rules as fraudulent but later verified by the card holders to be legitimate. Hence, the rules have to be adapted or augmented over time to capture all (but only) fraudulent transactions. In this paper, we present RUDOLF, a system whose goal is to assist users to define and refine rules for fraud detection.

Note that our goal resembles in part that of previous works on query/rule refinement, which attempt to automatically identify minimal changes to the query/rule in order to insert or remove certain items from the query result (e.g., [7]). However, a key difference here is that such minimal modifications often do not capture the actual “ground truth”, namely the nature of ongoing attack, which may not yet be fully reflected in the data. By interacting with users to fine-tune rules, important domain knowledge can be effectively imported into the rules to detect the pattern of frauds often even before they are manifested in the transactions themselves.

Our goal also resembles previous work on discovering or learning decision rules from streams with concept drifts (e.g. [8, 9]). Like them, RUDOLF strives to discover or adapt rules as new transactions arrive. However, all previous work considered only domains over numerical values and hence do not immediately apply to our setting, which involves both categorical and numerical values. Furthermore, RUDOLF makes crucial use of a hierarchy of higher-level concepts over the domains (numerical or categorical) in the specification of rules. In addition, RUDOLF collaborates with the domain expert to improve upon the quality of the rules used for capturing only fraudulent transactions. The interplay between the domain experts and the use of higher-level concepts, whenever possible, enables the derivation of rules which can be used to explain the true nature ongoing frauds. Our experiments indicate that such interactions can be effective in deriving rules with good prediction quality.

An overview example The top of Figure 1 shows a simplified set of rules that is currently used to capture fraudulent transactions up to yesterday. Intuitively, the first two rules capture a suspicion of two attacks on an online store taking place at the first and

last few minutes of 6pm, charging amounts over \$110. The last rule captures a fraud pattern at Gas station A where false charges of amounts over \$40 are made soon after the closing time at 9pm. In practice each rule also includes some threshold condition (not shown) on the score (i.e., the degree of confidence that the transaction is fraudulent) for each transaction, as well as additional conditions on the user/settings/etc. The scores and the additional conditions are omitted so that we can focus our discussions on the semantic aspect of the rules shown in Figure 1.

Figure 2 shows an example of a relation which contains a number of transactions made today. The transaction tuples are ordered by the time of the transaction. In the figure, some transactions that were reported as fraudulent are labeled as “FRAUD”. Similarly, transactions that are reported to be legitimate may be correspondingly labeled “LEGITIMATE” (not shown in the figure). Transactions may also be unlabeled. The current set of rules captures only the shaded tuples shown in the transaction relation. Clearly, none of the new fraudulent transactions are captured by the existing rules whereas some unlabeled transactions are captured.

RUDOLF first attempts to capture the fraudulent transactions by generalizing the rules, semantically according to a given ontology whenever possible, before it specializes the rules to avoid unnecessarily capturing legitimate transactions. However, the changes proposed by RUDOLF may not correspond to the best or correct changes. The domain experts can view/accept/reject/modify the suggestions provided by RUDOLF, arriving for instance at the

- 1) $\text{Time} \in [18:05, 18:05] \wedge \text{Amt} \geq 100 \wedge \text{Type} = \text{Onl.}, \text{ no CCV.}$
- 2) $\text{Time} \in [18:55, 19:15] \wedge \text{Amt} \geq 100 \wedge \text{Type} = \text{Onl.}, \text{ no CCV.}$
- 3) $\text{Time} \in [20:45, 21:15] \wedge \text{Amt} \geq 40 \wedge \text{Location} \leq \text{Gas Station} \wedge \text{Type} \leq \text{Offline.}$

Intuitively, the first two rules above flag online transactions without CCV, charging amounts over \$100 in the respective time intervals as fraudulent transactions. The third rule flags offline transactions at the gas stations around closing time of amounts over \$40 as fraudulent transactions. Observe that the condition “ $\text{Location} \leq \text{Gas Station}$ ” is a semantic generalization of Gas Stations A and B, which are defined in an ontology to be contained within the category “Gas Station”. Similarly, “ $\text{Type} \leq \text{Offline}$ ” reflects the semantic category (shown at the bottom of Figure 1) which contains offline transactions with and without PIN.

Contributions This paper makes the following contributions.

- (1) We formulate and present a novel interactive framework for determining the “best” way to adapt and augment rules so that fraudulent transactions are captured and, at the same time, legitimate transactions are avoided.
- (2) We establish that the rule refinement problem is NP-hard even under special circumstances: (1) determine the best way to generalize rules to capture new fraudulent transactions when there are no new legitimate transactions, and (2) determine the best way to specialize existing rules to avoid capturing new legitimate transactions when there are no new fraudulent transactions.
- (3) In light of these hardness results, we develop a heuristic algorithm which is able to interactively adapt rules with domain experts until a desired set of rules is obtained. At each step, the algorithm makes a proposal of the best changes to a rule, and the domain expert can further refine the proposed changes or seek suggestions for other possible changes. Our algorithm represents a departure from prior algorithms on discovering or learning decision rules from streams with concept drifts in that it handles categorical

values in addition to numerical values, adapt rules with semantic concepts from available ontologies, and interacts with domain experts.

- (4) We have implemented our solution in the RUDOLF prototype system and applied it on real data, demonstrating the effectiveness and efficiency of our approach. We performed experimental evaluations on a real-life dataset of credit card transactions. We show that by interacting with users (even ones with only little knowledge specific to the domain of the datasets), our algorithms consistently outperform alternative baseline algorithms, yielding more effective rules in shorter time.

While most of our exposition on the features of RUDOLF is based on credit card frauds, we emphasize that RUDOLF is a general-purpose system that can be used to interact with users to refine rules. For example, for preventing network attacks, for refining rules for spam detection or for intrusion detection [10].

A first prototype of the system was demonstrated at VLDB’16 [11]. The short paper accompanying the demonstration gives only a brief overview of the system architecture whereas the present work provides a comprehensive description of the underlying model and algorithms.

The paper is organized as follows. The next two sections define the model and problem statement behind RUDOLF (Section 2 and, respectively, Section 3). The algorithm behind RUDOLF is described in Section 4. We then present our experimental results (Section 5) and related work (Section 6), before we present our conclusions (Section 7).

2 PRELIMINARIES

Transaction relation A *transaction relation* is a set of tuples (or *transactions*). The transaction relation is appended with more transactions over time. We assume that the domain of every attribute A has a partial order, which is reflexive, antisymmetric, and transitive, with a greatest element \top_A and least element \perp_A . W.l.o.g. we also assume that \perp_A does not appear in any of the tuples¹. For brevity, when an attribute name is clear from the context we will omit it and simply use the notations \top and \perp . Attributes that are associated with a partial order but not a total order are called *categorical attributes*. The elements in such partial order are sometimes referred to as *concepts*.

A transaction may be flagged as *fraudulent* which means that the transaction was carried out illegally or conversely, a transaction may be flagged as *legitimate*. Unmarked transactions are called *unlabeled* transactions. The labeling is assumed to correspond to the (known part of the) *ground truth*. In addition, each transaction has a *score* between 0 and 1, that is computed automatically using machine learning techniques, and depicts the estimated probability of each transaction to be fraud. The score may or may not agree with the ground truth and this discrepancy is precisely the reason why rules are employed to refine the fraud detection.

Example 2.1. Part of a transaction relation I with schema $T(\text{time}, \text{amount}, \text{type}, \text{location}, \dots)$ is shown in Figure 2. Each tuple records, among others, the time, amount, type of transaction, and location where the purchase was made through some credit card. The scores of the transactions, as computed by a machine learning module, are omitted from the figure. The last column annotates the type of transactions. The part of instance I that is shown contains only fraudulent and unlabeled transactions.

¹If this is not the case, add a new special element to the domain and set it smaller, in the partial order, than all other elements.

Existing fraud rules Φ from the previous day:

- 1) $\text{Time} \in [18:00, 18:05] \wedge \text{Amt} \geq 110$
- 2) $\text{Time} \in [18:55, 19:00] \wedge \text{Amt} \geq 110$
- 3) $\text{Time} \in [21:00, 21:15] \wedge \text{Amt} \geq 40 \wedge \text{Location} \leq \text{'Gas Station A'}$

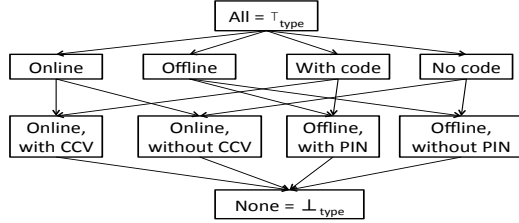


Figure 1: Top: An existing set of rules. Bottom: Partial Order for type values.

The type `time` is discretized to minutes and is associated with a date (not shown). It thus has a partial order (in fact, a total order), given by the \leq relation. The type `amount` also has a total order with least element 0 and greatest element ∞ . The attribute `type` is a categorical attribute and its partial order is given by the hierarchy shown at the bottom of Figure 1. Some examples of concepts in the hierarchy are “Online” or “Offline, without PIN”. The type `location` is also a categorical attribute and its partial order (not shown) is given by, say, the geographical containment relationship. In particular, “Gas Station A” and “Gas Station B” are both concepts that are children of the concept “Gas Station”.

Rules For simplicity and efficiency of execution, we assume rules are typically written over a single relation, which is a universal transaction relation that includes all the necessary attributes (possibly aggregated or derived from many other database relations) for fraud detection. Hence, it is not necessary to consider explicit joins over different relations in the rule language.

To highlight the key principles of our approach we consider here a fairly simple rule language that captures a disjunction of conjunctions. For simplicity, our rule language allows only one condition over each attribute. Multiple disjunctive conditions over the same attribute can be expressed in multiple rules. Other extensions to the rule language are possible but will not be considered here. Note that the rule language that we consider, albeit simple, forms the core of common rule languages used by actual systems.

A *rule* is a conjunction of one or more conditions over the attributes of the transaction relation. More precisely, a rule is of the form $\alpha_1 \wedge \dots \wedge \alpha_n$ where n is the arity of the transaction relation, α_i is a condition is of the form ‘ $A_i \text{ op } s$ ’ or ‘ $A_i \in [s, e]$ ’, A_i is the i th attribute of the transaction relation, $\text{op} \in \{=, <, >, \leq, \geq\}$, and s and e are constants.

More formally, if φ is a rule that is specified over a transaction relation I , then $\varphi(I)$ denotes the set of all tuples in I that satisfy φ . We say that $\varphi(I)$ are the transactions in I that are *captured* by φ . If Φ denotes a set of rules over I , then $\Phi(I) = \bigcup_{\varphi \in \Phi} \varphi(I)$. In other words, $\Phi(I)$ denotes the union of results of evaluating every rules in Φ over I . Observe that $\Phi(I) \subseteq I$ since every rule selects a subset of transactions from I . For readability, in our examples we show only the non trivial conditions on attributes, namely omit conditions of the form $A_i \leq \top$.

Note that, for simplicity, each rule includes only one condition over each attribute, but multiple disjunctive conditions over the same attribute can be expressed using multiple rules.

Time	Amount	Transaction Type	Location	
18:02	107	Online, no CCV	Online Store	FRAUD
18:03	106	Online, no CCV	Online Store	FRAUD
18:04	112	Online, with CCV	Online Store	
19:08	114	Online, no CCV	Online Store	FRAUD
19:10	117	Online, with CCV	Online Store	
20:53	46	Offline, without PIN	GAS Station B	FRAUD
20:54	48	Offline, without PIN	GAS Station B	FRAUD
20:55	44	Offline, without PIN	GAS Station B	FRAUD
20:58	47	Offline, with PIN	Supermarket	
21:01	49	Offline, with PIN	GAS Station A	
:	:	:	:	:

Figure 2: A transaction relation containing new transactions.

Example 2.2. The top of Figure 1 illustrates a set Φ of three simplified rules currently used by the example credit card company to detect fraudulent transactions. The first rule captures all transactions made between 6pm to 6:05pm where the amount involved is at least \$110. As previously mentioned, in practice each rule also includes some threshold conditions (not shown here) on the score for each transaction, as well as additional conditions on the user/settings/etc. For simplicity we omit the score thresholds and the additional conditions and focus in the sequel on the simplified rules in this example.

For the new transaction relation shown in Figure 2, this rule captures the 3rd tuple (which is an unlabeled transaction). The 2nd rule captures no tuples, and the 3rd rule captures the 10th unlabeled tuple. Hence, the existing rules Φ do not capture any of the fraudulent transactions on the current day.

As we shall demonstrate, the rule language that we consider here, even though simple, is able to succinctly capture the fraudulent transactions (and avoid legitimate tuples) in our experiments with a real dataset. Other domains rules, e.g. access control for network traffic or spam detection rules can also be expressed in our language.

Cost and Benefit of modifications A *modification* to a rule is a change in a condition of an attribute in the rule. One may also *copy* an existing rule before modifying the copy, add rule or remove an existing rule. As we will elaborate in subsequent sections, our cost model assumes there is a cost associated with every operation/modification made to a condition in the rule.

To compare between possible modifications to rules and determine which modifications are better, we need to know not only the cost of each modification but also the “benefit” it entails. Intuitively, the gain from the modifications can be measured in three ways: (1) the increase in the number of fraudulent transactions that are captured by the modified rule, (2) the decrease in the numbers of legitimate transactions that are captured by the modified rule, and (3) the decrease in the numbers of captured unlabeled transactions. The assumption underlying (3) is that unlabeled transactions are typically assumed to be correct until explicitly reported/tagged otherwise and the more specific the rules are to fraudulent (and only) fraudulent transactions, the more precise they are in embodying possible domain-specific knowledge about the fraudulent transactions.

Observe that if our modifications are ideal, then after the modifications, there are more fraudulent transactions captured, and less

legitimate and unlabeled transactions caught. Subsequently, the overall update cost is defined as the cost of modifications minus their benefit. We give the formal definition in the next section.

3 PROBLEM STATEMENT

As described, the goal of RUDOLF is to identify minimal modifications to existing rules that would ideally capture all fraudulent transactions, omit all legitimate transactions, and at the same time, minimize the inclusion of unlabeled transactions. The modifications suggested by RUDOLF serve as a starting point for the domain expert who can either accept the proposed modifications or interactively refine them with the help of RUDOLF. Formally, the problem is stated below.

Definition 3.1 (General Rule Modification Problem). Let Φ be a set of rules on an existing transaction relation I . Let I' denote a new set of transactions over the same schema. Let $F, L \subseteq I$ (resp. $F', L' \subseteq I'$) be two disjoint sets of *fraudulent* and *legitimate* transactions in I (resp. I'). Let $R = I - (F \cup L)$ (resp. $R' = I' - (F' \cup L')$) be the remaining *unlabeled* transactions in I (resp. I').

The GENERAL RULE MODIFICATION PROBLEM is to compute a set M of modifications to Φ to obtain Φ' so that $\text{cost}(M) - (\alpha * \Delta F + \beta * \Delta L + \gamma * \Delta R)$ is minimized, where $\alpha, \beta, \gamma \geq 0$, and

- $\Delta F = |(F \cup F') \cap \Phi'(I \cup I')| - |(F \cup F') \cap \Phi(I \cup I')|$,
- $\Delta L = |(L \cup L') \cap \Phi(I \cup I')| - |(L \cup L') \cap \Phi'(I \cup I')|$, and
- $\Delta R = |(R \cup R') \cap \Phi(I \cup I')| - |(R \cup R') \cap \Phi'(I \cup I')|$.

The term $(\alpha * \Delta F + \beta * \Delta L + \gamma * \Delta R)$ represents the benefit of applying the given modifications to the rules. In our rule language, if a fraudulent transaction is not captured by a set of rules, then at least some condition of a rule needs to be generalized to select that fraudulent transaction. At the same time, making a condition more general may capture also legitimate or unlabeled transactions. Conversely, if a legitimate transaction is captured by a set of rules, then at least some condition of a rule needs to be made more restrictive so that the legitimate transaction is excluded. Such modifications carry the risk of omitting some fraudulent transactions that should be captured. The coefficients α, β and γ are non-negative and are typically provided by the user to tune the relative importance of each category (resp. correctly capturing the fraudulent transactions, avoiding misclassifying legitimate transactions, and excluding unlabeled transactions) in the calculation of benefit. The overall goal is to identify the modifications having the best cost-benefit balance.

Observe that if α, β and γ are set to large numbers (e.g. greater than the maximal update cost) then the most beneficial modifications are those leading to a “perfect” set of rules, namely one that will (1) capture all fraudulent transactions, (2) exclude all legitimate transactions, and at the same time, (3) does not capture any unlabeled transactions.

4 THE GENERAL RULE MODIFICATION ALGORITHM

The rule modification algorithm (outlined below) first interactively refines the rules to capture fraudulent transactions. The expert can stop the refinement whenever she is satisfied with the rules and believes that the omission of the remaining fraudulent transactions is tolerable. The resulting set of rules may capture some (existing) legitimate tuples. Hence, in the second step, the algorithm continues to interactively refine the rules to avoid the legitimate transactions. Here again the user may stop the algorithm when she believes that the inclusion of the remaining legitimate transactions

is acceptable. However, the rules that result after this step may no longer capture some fraudulent transactions that were previously captured. The domain expert can either repeat the process described above to remedy this, or choose to end the rule refinement process at this point. In the latter case, the domain expert has a choice to leave the result as-is or allow the algorithm to create transaction-specific rules to capture each of the remaining transactions.

- (1) Generalize rules to capture fraudulent transactions. See Algo. 1, section 4.1.
- (2) Specialize rules to avoid legitimate transactions. See Algo. 2, section 4.2.
- (3) Exit if the domain expert is satisfied. Otherwise, repeat the steps above.

Observe that it is essential for the generalization algorithm (Algo. 1) to be applied before the specialization algorithm (Algo. 2) as one can always add rules to capture specific fraudulent transactions without accidentally capturing legitimate transactions. On the other hand, one cannot always add/modify rules to avoid specific legitimate transactions without accidentally excluding fraudulent transactions with our current rule language.

As we shall show in the next sections, finding an optimal set of changes to the rules is computationally expensive in either case. For this reason, instead of computing an optimal set of modifications to the rules to generalize or special rules to capture fraudulent and, respectively, avoid legitimate tuples, we develop heuristic algorithms to identify the best update candidates in each of the cases.

4.1 Rule Generalization Algorithm

We first consider how rules can be generalized to capture fraudulent transactions when the set I' of new transactions contains only fraudulent transactions. The goal here is to adapt the existing set of rules Φ to capture all fraudulent transactions. We call this problem the RULE GENERALIZATION PROBLEM

THEOREM 4.1. *The RULE GENERALIZATION PROBLEM is NP-hard even if Φ is perfect for I , (namely, $F \subseteq \Phi(I)$ and $L \cap \Phi(I) = R \cap \Phi(I) = \emptyset$). The problem is NP-hard even when I contains only unlabeled transactions and I' consist of only one fraudulent transaction.*

PROOF. We prove the two claim simultaneously by reduction from the minimum hitting set problem which is known to be NP-hard. We recall of the Minimum Hitting Set Problem below.

Definition 4.2 (Minimum Hitting Set). Consider the pair (U, S) where U is a universe of elements and S is a set of subsets of U . A set $H \subseteq U$ is a hitting set of S if H hits every set in S . In other words, $H \cap S' \neq \emptyset$ for every $S' \in S$. A minimum hitting set H is a minimum cardinality hitting set s.t. $\forall e \in H$, we have $H \setminus \{e\}$ is not a hitting set of S .

We assume that each rule modification is associated with a unit cost. Given an instance of the hitting set problem, we construct an instance of the RULE GENERALIZATION PROBLEM were there no fraudulent or legitimate transactions in I and I' consist of only one fraudulent transaction, as follows.

The transaction relation has $|U|$ columns, one for each element in U . The transaction relation I has a characteristic (unlabeled) tuple for every set $s \in S$. That is, for every set s in S , we construct a characteristic tuple of s in I by placing a 0 in position i if $x_i \in s$ and 1 otherwise. Hence there are $|S|$ tuples in the transaction

relation I and these are unlabeled tuples that we would like to minimize capturing with the rules. There are no existing fraudulent or legitimate tuples in I . The set Φ is initially empty. Hence, $\Phi(I)$ does not capture any transaction (and thus, by definition, is “perfect” for D). The instance I' consists of a single transaction $(1,1,1,\dots,1)$, which is the new fraudulent transaction that we wish to capture.

As an example, consider the following hitting set where $U = \{A_1, A_2, A_3, A_4, A_5\}$ and $S = \{s_1, s_2, s_3\}$, where $s_1 = \{A_1, A_2, A_3\}$, $s_2 = \{A_2, A_3, A_4, A_5\}$, and $s_3 = \{A_4, A_5\}$. The transaction relation $I \cup I'$ (where I' is highlighted in gray) is shown below. The last column annotates the type of tuple (e.g., the last tuple is a fraudulent transaction).

A ₁	A ₂	A ₃	A ₄	A ₅	
0	0	0	1	1	
1	0	0	0	0	
1	1	1	0	0	
1	1	1	1	1	F'

It is straightforward to verify that the construction of the instance of the Rule Modification Problem can be achieved in polynomial time in the size of the input of the Hitting Set Problem. We now show that a solution for the Minimum Hitting Set Problem implies a solution for the Rule Modification Problem and vice versa.

Let H be the minimum hitting set. For every element x_i in H , we add a condition $a_i = 1$ to the same rule. (The condition can also be $a_i \geq 1$ or $a_i > 0$ but wlog we assume $a_i = 1$ is used.) The cost of changing $a_i = 1$ is 1. Clearly, we have that $\Phi'(I \cup I')$ contains the fraudulent transaction and since H hits every set $s \in S$, for every $s \in S$, there must be at least an attribute a_i in the corresponding tuple of s whose value 0 when the condition $a_i = 1$ according to the new rule Φ' . Hence, it follows that $\Phi'(I \cup I')$ does not contain any tuples in $I \cup I'$ other than the fraudulent tuple.

We show next that the expression $\text{cost}(M) - (\alpha * \Delta F + \beta * \Delta L + \gamma * \Delta R)$ is the minimum possible, assuming that $\alpha = \beta = \gamma > 1$ (β is actually irrelevant here) and the cost of each modification is 1. Suppose there is another set of modifications whose cost is lower than the above. It must be that none of the unlabeled tuples are selected by the modified rules since one can always avoid capturing an unlabeled tuple and lower the cost even further by adding an appropriate condition $a_i = 1$. Furthermore, by the same reason, the fraudulent tuple must be selected by the modified rules. Thus, the expression $\text{cost}(M) - (\alpha * \Delta F + \beta * \Delta L + \gamma * \Delta R)$ is the minimum possible when the number of $a_i = 1$ conditions correspond to the size of a the minimum hitting set. Any smaller value will mean we have a smaller cardinality hitting set which is a contradiction to our assumption that H is a minimum hitting set.

For the converse, let M be the set of modifications made to Φ such that $\text{cost}(M) - (\alpha * \Delta F + \beta * \Delta L + \gamma * \Delta R)$ is minimum. Again, wlog, we may assume that the modifications must be of the form $a_i = 1$.

Let $H = \{x_i \mid a_i = 1 \text{ in the modified rule}\}$. We show next that H is a minimum hitting set. First, we show that H is a hitting set. Suppose not, then there is a set $s \in S$ such that $H \cap s = \emptyset$. Let t be the tuple that corresponds to s in the transaction table. This means that $\Phi'(I \cup I')$ contains t , since t contains the value 1 in every attribute a_i where $a_i = 1$ in the modified rule. Pick an element, say $x_j \in s$ such that $x_j \notin H$. Now if we add the modification $a_j = 1$, the change in cost is $+1 - \gamma$. Since $\gamma > 1$, we have that the new total cost is lower than the original cost which contradicts the assumption that M is a set of modifications that would give the minimum total cost.

Next, we show that H is a minimum hitting set. Suppose not, then there is another hitting set H' where $|H'| < |H|$. With H' , it is straightforward to construct a set of modifications whose cost is lower than $\text{cost}(M) - (\alpha * \Delta F + \beta * \Delta L + \gamma * \Delta R)$.

In our running example, a modified rule is

$$A_1 \leq \top \wedge A_2 = 1 \wedge A_3 \leq \top \wedge A_4 = 1 \wedge A_5 \leq \top$$

since a minimum hitting set is $\{A_2, A_4\}$. \square

The reduction of the above proof shows that the NP-hardness result may arise because we allow the size of the schema of the transaction relation to vary. We show next that, even if we fix the size of the schema, the NP-hardness result continues to hold.

THEOREM 4.3. *The RULE GENERALIZATION PROBLEM is NP-hard even if Φ is perfect for I and the size of the schema of the transaction relation is fixed.*

SKETCH. The proof makes use of a reduction from the set cover problem and in the reduction, a single unary transaction relation is used. We build a taxonomy of the elements of a set cover instance according to which element belongs to which set. The relation is initially empty and assume Φ is initially empty as well. The cost of adding rule with a condition is 1 and we assume that the cost of adding the condition $A \leq \top$ is very high (i.e., it is prohibited). The new transaction relation I' consists of n new fraudulent transactions, one for each element of the universe in the set cover instance. One can then establish that a set of rules of minimum cost can be derived from a minimum set cover and vice versa. Intuitively, each rule has the form $A \leq S_i$ where each S_i is part of the solution to the instance of the minimum set cover problem. \square

The Algorithm In view of the hardness result, we develop a heuristic algorithm (Algo. 1) for determining a best set of generalizations to capture a given set of fraudulent transactions.

In the algorithm, we use I to denote both old and new transactions. Observe that one reason for the hardness of the rule generalization problem comes from the desire to identify a set of modifications that captures all fraudulent transactions and is *globally optimal*. Instead, our heuristic works in a greedy manner, gradually covering more and more uncaptured transactions. Rather than treating each transaction individually, we split the fraudulent transactions into smaller groups (clusters) of transactions that are similar to each other, based on a distance function, and treat each cluster individually. We denote the set of clusters by C . Each cluster in C is represented by a *representative tuple*. Intuitively, a representative tuple of a cluster is a tuple that “contains” every tuple in that cluster. Hence, if a rule is generalized to capture the representative tuple, then it must also capture every tuple in the associated cluster. The algorithm then identifies for each representative tuple the (top-k) best rule modifications to capture it. The proposed modifications are verified with the domain expert, who may interactively further adapt them or ask for additional suggestions. Note that the modifications made to the rules by the algorithm may result in capturing some legitimate tuples. We will see how this too may be avoided later.

We next describe our algorithm more formally. We first define each of the components, then provide a comprehensive example that illustrates all.

Representative tuple of a cluster The *representative tuple* f of a cluster C is a tuple with the same schema as tuples in C such that for every attribute A , $f.A$ contains $t.A$ for every $t \in C$. If A is an attribute with a total order, then $f.A$ is an interval that contains $t.A$.

Algorithm 1: Generalize rules to capture new fraudulent tuples

Input: A set Φ of rules for a transaction relation I (contains old and new transactions), with $F \subseteq I$ and $F' \subseteq I$.

Output: A new set Φ' of rules that captures $F \cup F'$.

```
1 Let  $C$  denote the result of clustering tuples in  $F \cup F'$ .
2 foreach  $C \in C$  do
3   Let  $f(C)$  be the representative tuple of  $C$ .
4   Let  $\text{Top-}k(f(C))$  denote the top- $k$  rules for  $f(C)$  based on
   Equation 2.
5 foreach  $C \in C$  do
6   while there does not exist a rule  $r$  such that  $f(C) \in r(I)$  do
7     if  $\text{Top-}k(f(C))$  is non-empty then
8       Remove the next top rule  $r$  from  $\text{Top-}k(f(C))$ .
9       Construct the smallest generalization of  $r$  to  $r'$  so
       that  $f(C) \in r'(I)$ .
10      Ask whether the rule  $r'$  is correct.
11      if the domain expert agrees with the modified  $r'$  then
12        Replace  $r$  with  $r'$  in  $\Phi$ .
13      else
14        Ask the domain expert which modifications in  $r'$ 
        are undesired.
15        Revert the modifications to the original
        conditions of  $r$  as indicated by the domain
        expert.
16        Allow the domain expert to make further
        generalizations to the proposed rule.
17      else
18        Create a rule that will select exactly  $f(C)$  and add it
        to  $\Phi$ .
19 return  $\Phi$  as  $\Phi'$ .
```

If A is a categorical attribute, then $f.A$ is a concept that contains $t.A$ for every $t \in C$. Furthermore, $f.A$ is the smallest interval (resp. concept) that has the above property.² In other words, f is the “smallest” such tuple that contains every tuple in C . Intuitively, the clustering step, which generates representative tuples, provides a higher-level semantic abstraction to the fraudulent tuples that are to be captured.

Distance of a rule from a representative tuple The notation $|f - r|$ denotes the *distance* of a rule r from a representative tuple f . Intuitively it reflects how much the conditions in the rule need to be generalized for the rule to capture the representative tuple. It is formally defined as:

$$\sum_i^n (f.A_i - r.A_i), \quad (1)$$

where $r.A_i$ denotes the interval or concept associated with the condition of attribute A_i in the rule r , and n is the arity of the transaction relation.

The distance between two attribute intervals is defined as follows. If $f.A$ is the interval $[s_1, e_1]$ and $r.A$ is the interval $[s_2, e_2]$, then $|[s_1, e_1] - [s_2, e_2]|$ is the sum of sizes of the smallest interval(s) needed to extend $[s_2, e_2]$ so that it contains $[s_1, e_1]$. For example, the distance of $|[1, 5] - [5, 100]|$ is 4, while the distance of $|[1, 100] - [1, 5]|$ is 95. The distance of $|[5, 10] - [1, 100]|$ is 0, since $[1, 100]$ already covers $[5, 10]$. If an attribute A is categorical, then $|f.A - r.A|$ is the length of the smallest “ontological distance” that need to be added to $r.A$ so that it contains $f.A$. For example, $|\text{Offline with PIN - Online with CCV}|$ is 1, and $|\text{Offline without}$

$\text{PIN - Online with CCV}|$ is 2. By leveraging concepts in the ontology when available, the resulting rules have a more meaningful interpretation.

The overall cost function The overall cost of modifying a rule r to capture a representative tuple f then reflects the amount of modifications that need to be carried (Equation 1) minus the benefit derived from those modifications:

$$\sum_i^n (f.A_i - r.A_i) - (\alpha * \Delta F + \beta * \Delta L + \gamma * \Delta R) \quad (2)$$

Putting things together We now have all the ingredients for describing our algorithms. The algorithm proceeds by clustering the transactions into groups and then computes a representative tuple for each cluster.³ For every cluster C , we compute its representative tuple $f(C)$ as well the cost (according to Equation 2) of modifying each of the rules to capture it, and select the k rules with minimal cost. We refer to them as the top- k rules (see Line 4 of Algo. 1). In Line 8, we pick the top rule in $\text{top-}k(f(C))$. If the rule $r(I)$ does not already contain $f(C)$, we will attempt to make the smallest generalization on r to r' so that $r'(I)$ contains $f(C)$. Whenever the interval or concept of an attribute $r.A$ does not contain $f.A$, we will attempt to modify $r.A$ by computing the smallest extension needed on $r.A$ based on its distance from $f.A$. We perform this extension on every attribute A of r where $r.A$ does not contain $f.A$. This is what we mean by “generalize r minimally to r' so that $f(C) \in r'(I)$ ” in line 9.

Next, we proceed to verify the new rule r' with our potential modifications with the domain expert. If the domain expert agrees with the proposed modifications (lines 11,12), we will replace r with the new rule r' in Φ . Otherwise, we will refine our question to probe the domain expert further on whether or not there are parts of the modifications that are desired even though the entire rule r' is not what the domain expert desires (lines 14,15). We then modify only the desired modifications, if any. The next step allows the domain expert to make further generalizations to the rules. After this, the algorithm proceeds to pick another closest rule to $f(C)$ to attempt to capture $f(C)$. Line 18 captures the case when we ran out of rules to modify. If this happens, we will construct a new rule to cover f by stating the exact conditions that are required.

We conclude with a remark regarding the computational complexity of the algorithm. All components of the algorithm (i.e., clustering, computation of representative tuples for each cluster and the top- k rules) execute in PTIME in the size of its input. Hence each iteration executes in PTIME in the size of the input. The number of iterations per cluster is dependent on the amount of refinements that the expert makes to the suggested rule modifications (shown in our experiments to be fairly small).

Example 4.4. The relation below depicts the representative tuples of the clusters formed from the six fraudulent transactions from Figure 2. The first tuple is the representative tuple of the cluster that consists of the first two tuples in Figure 2. The second (resp. third) tuple below is the representative of the cluster that consists of only the 4th tuple in Figure 2 (resp., 6th, 7th, and 8th tuples).

²If there are multiple such concepts, e.g. in non tree-shaped concept hierarchies, we pick one.

³In our implementation, we use the clustering algorithms of [12], but other clustering algorithms can be similarly be used.

Representatives of fraudulent transactions in Figure 2:

Time	Amount	Transaction Type	Location
[18:02,18:03]	[106,107]	Online, no CCV	Online Store
[19:08,19:08]	[114,114]	Online, no CCV	Online Store
[20:53,20:58]	[44,48]	Offline, without PIN	GAS Station B
⋮	⋮	⋮	⋮

Consider a domain expert, Elena, that is working with the system. The first rule in Figure 1 is the closest to the first representative tuple above. This is because Equation 2 evaluates to $(0+4+0+0)-(2+0+0)=2$ for the first rule and the first representative tuple, whereas the second and third rule of Figure 1 and the first representative tuple have scores $(53+4+0+0)-(2+0-1)=56$ and $(178+0+0+1)-(6+0-3)=168$, respectively, and are thus ranked lower than the first rule. The number '1' in the last calculation denotes the ontological distance between "Gas Station A" and "Gas Station B". Since they are both contained under "Gas Station", the distance is 1.

Algo. 1 will thus propose to modify the condition of the first rule from "Amt ≥ 110 " to "Amt ≥ 106 " to capture the representative tuple. It then proceeds to verify the modification with Elena. Suppose Elena accepts the proposed modification but further generalizes the condition rounding it down to "Amt ≥ 100 " instead. So the new rule 1 is

- 1) $Time \in [18:00,18:05] \wedge Amt \geq 100$.

Besides the fact that rounded values may be preferred by domain experts over more specific values, such rounding may embody domain-specific knowledge that may possibly lead to the discovery of more fraudulent transactions, particularly from transactions that are unlabeled, or the discovery of legitimate transactions that should not have been labeled as legitimate.

For the second and third cluster, similar interactions occur between RUDOLF and Elena. The new rules that result are:

- 2) $Time \in [18:55,19:15] \wedge Amt \geq 110$.
- 3) $Time \in [20:45, 21:15] \wedge Amt \geq 40 \wedge Location \leq Gas\ station'$.

To conclude, observe that Algo. 1 allows Elena to make further generalizations to the rules. Elena rounded the value down from 106 to 100 because her experience tells her that if frauds occur with amount greater than \$106, then it is likely to occur a few dollars below \$106 as well. Hence, she generalized (i.e., rounded down) the value to \$100. In making such generalizations, the fraudulent transactions will continue to be captured. However, the modified rules may now capture (more) non-fraudulent transactions. Nonetheless, we still allow such generalizations since these are deliberate changes made by Elena, the domain expert. More typically, however, such "rounding generalizations" tend to be meaningful generalizations that may lead to the discovery of more fraudulent transactions (i.e., unlabeled transactions that should be classified as fraudulent or legitimate transactions that are mistakenly labeled as legitimate)⁴. As we shall show in Example 4.7, Elena can also leverage her experience or domain knowledge to pinpoint the right conditions for avoiding legitimate transactions. We describe next how over-generalization may be treated.

4.2 Rule Specialization Algorithm

In the previous subsection, we have seen how one generalizes rules to capture fraudulent transactions. We now discuss the opposite case, where we wish to specialize rules instead, in order to exclude legitimate transactions when there no new fraudulent transactions or unlabeled transactions but there are new legitimate transactions.

⁴This is from our conversations with domain experts on credit card fraud detection.

We call this special case the RULE SPECIALIZATION PROBLEM. Here again we can show hardness results analogous to Theorem 4.1 and 4.3.

THEOREM 4.5. *The RULE SPECIALIZATION PROBLEM is NP-hard even if Φ is perfect for I . The problem is NP-hard even when I contains only unlabeled transactions and I' consists of only one legitimate transaction.*

PROOF. Given an instance of the hitting set problem, we construct an instance of the RULE SPECIALIZATION PROBLEM as follows.

The transaction relation has $|U|$ columns, one for each element in U . For every set s in S , we construct a characteristic tuple of s by placing a 0 in position i if $x_i \in s$ and 1 otherwise. Hence there are $|S|$ tuples in the transaction relation I so far and the fraudulent transactions $F = I$. The set Φ consists of a single rule

$$A_1 \leq \top \wedge \dots \wedge A_{|U|} \leq \top,$$

where \top denotes the top element, and hence, $\Phi(I)$ currently captures all fraudulent transactions F . The new transaction relation I' consists of a single tuple $(1,1,\dots,1)$. This set L' of legitimate transactions is a singleton set consisting of only $(1,1,\dots,1)$. That is, $L' = I'$. This is the legitimate transaction that we wish to exclude.

With $F' = L = \emptyset$, our goal is to specialize the rule in Φ to capture exactly the fraudulent tuples F only. Like in the proof of Theorem 4.1, we assume that each modification is associated with a unit cost and $\alpha = \beta = \gamma > 1$.

As an example, consider the same hitting set as in the proof of Theorem 4.1, where $U = \{A_1, A_2, A_3, A_4, A_5\}$ and $S = \{s_1, s_2, s_3\}$, where $s_1 = \{A_1, A_2, A_3\}$, $s_2 = \{A_2, A_3, A_4, A_5\}$, and $s_3 = \{A_4, A_5\}$. The transaction relation $I \cup I'$ is shown below (where I' is shown in gray). The last column annotates the type of tuple (i.e., F for tuples in F and L' for tuples in L').

A ₁	A ₂	A ₃	A ₄	A ₅	
0	0	0	1	1	F
1	0	0	0	0	F
1	1	1	0	0	F
1	1	1	1	1	L'

It is straightforward to verify that the reduction to an instance of the RULE SPECIALIZATION PROBLEM can be achieved in polynomial time in the size of the input of the Hitting Set Problem. We now show that a solution for the Minimum Hitting Set Problem implies a solution for the RULE SPECIALIZATION PROBLEM and vice versa.

Let H be a minimum hitting set. For every element x_i in H , we duplicate the original rule (except if this is the last element in H) and modify the corresponding condition to $a_i = 0$ in the copy of the rule. (The condition can also be $a_i \leq 0$ or $a_i < 1$ but wlog we assume $a_i = 0$ is used.) Recall that the cost of changing $a_i = 0$ is 1, and the cost of duplicating a rule is 1. Clearly, we have that $\Phi'(I \cup I')$ contains F . Indeed, since H hits every set $s \in S$, there must be an element in s whose corresponding value under an attribute a_i is 0 when the condition $a_i = 0$ according to a new rule in Φ' . Hence, it follows that $\Phi'(I \cup I')$ contains F and since each rule in Φ' contains a condition of the form $a_i = 0$, the legitimate transaction $(1,1,\dots,1)$ will not be among $\Phi'(I \cup I')$.

We show next that the expression $\text{cost}(M) - (\alpha * \Delta F + \beta * \Delta L + \gamma * \Delta R)$ is the minimum possible. Suppose there is another set M' of modifications to Φ'' such that $\text{cost}(M') - (\alpha * \Delta F + \beta * \Delta L + \gamma * \Delta R)$ is less than the previous expression. Observe that every rule in Φ'' must contain at least a condition that is specific to selecting

a fraudulent transaction. That is, for every rule, $a_i = 0$ for some i since otherwise, the rule is either redundant or the legitimate transaction will be selected. Also, we can assume that every other condition in the rule in Φ'' cannot contain a condition that selects 1s (e.g., of the form $a_i = 1$). If a rule r contains a condition $a_i = 1$, then we can omit this condition and assume it is $a_i \leq \top$ instead. The rule with $a_i \leq \top$ captures all tuples that are captured by r (and possibly more) and hence, we will continue to capture all fraudulent tuples and continue to exclude the legitimate tuple under this assumption. Similarly, if a rule r contains multiple conditions $a_i = 0$ s, then we can omit all but one of the $a_i = 0$ s and assume the rest are $a_i \leq \top$. We can now construct a hitting set from M' that is smaller than H , which is a contradiction.

For the converse, let M be the set of modifications made to Φ such that $F \subseteq \Phi'(I \cup I')$, the legitimate transaction l is such that $l \notin \Phi'(I \cup I')$, and $\text{cost}(M) - (\alpha * \Delta F + \beta * \Delta L + \gamma * \Delta R)$ is minimum. As before, observe that each rule must contain at least one modification of the form $a_i = 0$ for some i so that l is not selected. Furthermore, it is easy to see that each rule must contain exactly one such condition $a_i = 0$ only as additional conditions such as $a_j = 1$ or $a_j = 0$ are redundant and can only increase the cost.

Let $H = \{x_i \mid a_i = 0 \text{ in any of the modified rules}\}$. We show next that H is a minimum hitting set. First, we show that H is a hitting set. Suppose not, then there is a set $s \in S$ such that $H \cap s = \emptyset$. In other words, for every element $x_i \in s$, there does not exist a rule in Φ' where $a_i = 0$. Let f be the tuple that corresponds to s in the transaction table. This means that $f \notin \Phi'(I \cup I')$, which contradicts our assumption that $F \subseteq \Phi'(I \cup I')$.

Next, we show that H is a minimum hitting set. Suppose not, then there is another hitting set H' where $|H'| < |H|$. With H' , it is straightforward to construct a set of modifications whose cost is lower than M 's cost.

In our running example, Φ' contains two rules:

$$\begin{aligned} A_1 \leq \top \wedge A_2 = 0 \wedge A_3 \leq \top \wedge A_4 \leq \top \wedge A_5 \leq \top \\ A_1 \leq \top \wedge A_2 \leq \top \wedge A_3 \leq \top \wedge A_4 = 0 \wedge A_5 \leq \top \end{aligned}$$

since a minimum hitting set is $\{A_2, A_4\}$. \square

Similarly, we can show that the NP-hardness continues to hold even if we fix the size of the schema.

THEOREM 4.6. *The RULE SPECIALIZATION PROBLEM is also NP-hard even if Φ is perfect for I and the size of the schema of the transaction relation is fixed.*

SKETCH. The proof of the above result is similar to that of Theorem 4.3. It makes use of a reduction from the set cover problem and in the reduction, a single unary transaction relation is used. We build a taxonomy of the elements of a set cover instance according to which element belongs to which set. The relation initially contains all elements of the universe of the set cover instance and these transactions are all fraudulent. The set Φ consists of a single rule $A \leq \top$ which captures all fraudulent transactions. The cost of adding a rule and modifying a condition cost 1 each. The new transaction relation I' consists of a single legitimate tuple whose value does not occur among the existing values. One can then establish that a set of rules of minimum cost can be derived from a minimum set cover and vice versa. Intuitively, each rule has the form $A \leq S_i$ where each S_i is part of the solution to the instance of the minimum set cover problem. \square

Algorithm 2: Adapt rules to exclude legitimate tuples

Input: A set Φ of rules for a transaction relation I (contains old and new transactions) with $L \subseteq I$ and $L' \subseteq I$.

Output: A new set Φ' of rules that excludes $L \cup L'$.

```

1  foreach  $l \in (L \cup L')$  do
2      Let  $\Omega_l = \{r \in \Phi \mid l \in r(I)\}$ .
3      foreach  $r \in \Omega_l$  do
4          repeat
5              Let  $A$  be an attribute that has not been considered
6              before and where splitting on  $A$  to exclude  $l.A$  will
7              minimize the cost associated with splitting on  $A$ .
8              Suppose the existing condition on  $A$  is  $A \in [b, e]$ .
9              Split  $r$  into  $r_1$  and  $r_2$  on  $A$  as follows:
10             Let  $r_1$  be a copy of  $r$  except that the condition on  $A$ 
11             is  $A \in [b, \text{prev}(r.A)]$ .
12             Let  $r_2$  be a copy of  $r$  except that the condition on  $A$ 
13             is  $A \in [\text{succ}(r.A), e]$ .
14             Ask the domain expert whether the split into  $r_1$  and
15              $r_2$  is correct.
16             if the domain expert agrees with the modification
17             then
18                 Add  $r_1$  and  $r_2$  to  $\Phi$ .
19                 Allow the domain expert to make further
20                 modifications to the proposed rules.
21                 Break out of repeat loop.
22             until all attributes have been considered;
23             Remove  $r$  from  $\Phi$ .
24  return  $\Phi$  as  $\Phi'$ .
```

The Algorithm In view of the hardness results, we develop a heuristic algorithm (Algo. 2) that greedily determines the best attribute to “split” to avoid capturing each legitimate tuple.

In Algo. 2, we use I to denote both old and new transactions. For each legitimate transaction l in $(L \cup L')$, we determine the set Ω_l of rules that will capture l and modify every rule in Ω_l to ensure that the modified rules will no longer capture l . For a rule r in Ω_l , the algorithm proceeds to pick an attribute A where we can split the condition of the attribute to exclude the value $l.A$. The attribute A that we pick is the one that maximizes the benefit according to the benefit $\alpha * \Delta F + \beta * \Delta L + \gamma * \Delta R$, assuming a fixed cost of modification where we copy the rule and split on the attribute. If there are multiple attributes with the same maximum benefit, we randomly pick one of them. Observe that the heuristic of greedily selecting an attribute that will maximize benefit may not be globally optimal in the end. In the proof of Theorem 4.5, this greedy heuristic is analogous to the strategy of repeatedly picking the attribute (and splitting on the attribute) that will “hit” the most sets until all sets are hit.

Splitting on attributes Once an attribute is selected, the rule is duplicated into r_1 and r_2 and the condition on A in both rules is modified to exclude $l.A$. Observe that since $r(I)$ captures l , we must have that $l.A$ satisfies the rule r 's condition on A . In the split, r_1 's condition on A accept values from b to the element that is the predecessor of $l.A$, where b denotes the smallest value accepted by the existing condition of r on A . The rule r_2 selects only elements from the successor element of $l.A$ to the largest value (i.e., e) accepted by the existing condition of r on A .

For domains that are discrete and has a total order, the above procedure, $\text{prev}(r.A)$ and $\text{succ}(r.A)$ are well-defined. However, when domains are categorical and has only a partial order, the rules will be split according to the partial order. Let O denote the

set of all concepts (excluding $l.A$) that are parents of \perp in the partial order (i.e., the leaf nodes of the partial order excluding \perp). To exclude $l.A$, the algorithm considers how to select a minimum set of concepts to “cover” all concepts in O that excludes $l.A$ at the same time. It can be shown that the problem of computing such a minimum set is analogous to computing a minimum set cover for O . Our procedure adopts the greedy heuristic where we greedily pick a concept in the partial hierarchy that covers the most number of uncovered concepts in O until all nodes in O are covered. For categorical attributes, it may be necessary to duplicate r more than twice, where there is a rule to select each concept in the cover. For example, referring to Figure 1, to exclude “Online, with CCV”, we may pick “Offline” and “Online, without CCV” to cover the remaining concepts that are parents of “None”. Observe that similar to our algorithm on rule generalization, our rule specialization algorithm also makes use of the ontology whenever available to split the attributes meaningfully. In this case, attributes are split into meaningful concepts in the “lower-level” according to the ontology.

After this, we ask whether the domain expert agrees with the split. If the domain expert agrees, we add both rules r_1 and r_2 to Φ . The domain expert can also add further modifications to the rules (line 13), such as excluding more values than what is suggested by the algorithm, and we break out of the repeat loop. Otherwise, we repeat the loop to attempt to split r on another attribute to avoid capturing l . Note that since l has to be excluded, one of the splits must be deemed correct by the domain expert. After this, we remove r from Φ and repeat the same procedure to modify Φ to exclude the selection of another legitimate transactions.

Example 4.7. We will now illustrate Algo. 2. The legitimate transactions from Figure 2 that are captured by the modified rules of Example 4.4 are shown below for convenience.

	Time	Amount	Transaction Type	Location
l_1	18:04	112	Online, with CCV	Online Store
l_2	19:10	117	Online, with CCV	Online Store
l_3	21:01	49	Offline, with PIN	GAS Station A
	:	:	:	:

Modified rules Φ from Example 4.4:

- 1) $Time \in [18:00,18:05] \wedge Amt \geq 100$.
- 2) $Time \in [18:55,19:15] \wedge Amt \geq 110$.
- 3) $Time \in [20:45, 21:30] \wedge Amt \geq 40 \wedge Location \leq Gas\ station'$.

We would like to adapt the rules to exclude these legitimate transactions (and still continue capture fraudulent transactions). For this example, assume that $\alpha = \beta = \gamma = 1$.

The algorithm considers every legitimate transaction. Since l_1 is caught by rule (1) above, Algo. 2 proceeds to determine which attribute of the rule to split in order to exclude l_1 . Splitting on `time` or `amount` or `type` will result in the same maximum benefit: $(1*0$ (zero unlabeled transactions on either day) $+ 1*0$ (the number of fraudulent transactions that are caught remains unchanged) $+ 1*1$ (one less legitimate transaction that is caught)). Splitting on the attribute `location`, however, will cause additional fraudulent transactions (i.e., the first two transactions in Figure 2) to be missed and hence has a lower benefit than the rest of the attributes.

Suppose the algorithm proposes to split on `time` (an arbitrary choice among `time`, `amount` or `type`). This will result in two rules that will capture all fraudulent transactions that were previously caught by the rule and exclude l_1 at the same time.

- $$r_{11}: Time \in [18:00,18:03] \wedge Amt \geq 100.$$
- $$r_{12}: Time \in [18:05,18:05] \wedge Amt \geq 100.$$

At this point, Elena can accept this proposal or ask for alternatives. For the purpose of illustrating our algorithm, suppose Elena asked for an alternative proposed modification. Our algorithm may now propose to split on `type` instead. Since there is currently no condition on `type` in the first rule, the condition is implicitly “`type \leq T`”. And because the concepts “Offline” and “Online, without CCV” cover all possible type values (i.e., values immediately above the “None” node in Figure 1) except “Online,with CCV” which we wish to exclude, we have the following two rules:

- $$r_{11}: Time \in [18:00,18:05] \wedge Amt \geq 100 \wedge Type \leq Offline.$$
- $$r_{12}: Time \in [18:00,18:05] \wedge Amt \geq 100 \wedge Type \leq Onl, no CCV.$$

Using domain knowledge that only online purchases, especially those without CCVs are of concern, Elena eliminates the rule r_{11} .

After this, our algorithm proceeds in a manner that is similar to what was described before to split the second rule of Φ to omit l_2 (and similarly, the third rule of Φ for l_3). We omit the details here but show the final rules that are obtained.

- $$r_{22}: Time \in [18:55,19:15] \wedge Amt \geq 100 \wedge Type \leq Onl, no CCV.$$
- $$r_{31}: Time \in [20:45,21:15] \wedge Amt \geq 40 \wedge Location \leq Gas Station \wedge Type \leq Online.$$
- $$r_{32}: Time \in [20:45,21:15] \wedge Amt \geq 40 \wedge Location \leq Gas Station \wedge Type \leq No code.$$

Observe that whenever a condition is generalized (in Algo. 1), more legitimate or unlabeled tuples may be inadvertently captured by the rule. Hence, further refinements may be needed to tune the rules to a desired state. Conversely, if a condition is specialized, some fraudulent tuples may be inadvertently omitted. Hence, further refinements may be needed to tune the rules to a desired state. As we shall describe next, the rules are interactively refined based on the input of a domain expert such as Elena. In particular, there may be several rounds of refinements through generalizations and specializations before a desired set of rules is obtained.

5 IMPLEMENTATION AND EXPERIMENTS

RUDOLF is implemented in PHP/JavaScript and uses MySQL as the DB engine. Detailed system architecture described in [11].

Datasets We have access to a real-world datasets of credit card transactions by a financial company XYZ⁵. Due to the sensitivity of credit card-related information, we used anonymized version of the dataset. The dataset consists of transaction sets of various sizes from 15 financial institutes (FIs) for the first quarter of 2016. Each transaction set varies from 100K to 10M transactions and most of them consists of about 500K transactions. The percentage of fraudulent transactions varies between 0.5% to 2.5% between different FIs. The number of misclassified transactions (i.e., fraudulent transactions that are marked as legitimate and vice-versa) varies between 35% and 50%. The transactions contain both numerical (time, amount, number of previous actions, etc.) and also categorical (location, client type, etc) data. Along with the transactions, we obtained 15 rules-sets, one for each of the 15 FIs for the same time period from company XYZ. We also obtained the change history and versions of those rules. A small FI typically has about 10 rules while a big FI typically has about 130 rules. Most FIs have about 55 rules on average. Each time the rules are modified, the rules undergo about 10 rounds of modifications on average. The transactions in the data sets are annotated as fraudulent/legitimate, and we take these annotations as the ground truth. Each transaction also has a risk score, which is a value between 0 and 1000, that is generated by the company’s machine learning algorithm to determine the chance that the transaction is fraudulent. The

⁵Name omitted per company request

fraudulent transactions can be captured by the set of rules given by the company and allowing the users to refine the rules over time. Another option is to apply a rule that classify all transactions with risk score above a certain threshold as fraudulent.

Ontology In the experiments for the location attributes we used a geographical ontology (containing different relations, e.g., capital city, located in, region, continent, etc) that was built semi-automatically (using DBPedia [13]) and manually verified by the domain experts.

Experiment scenarios The different sizes of transaction sets allowed us to vary our experiments with different dataset sizes (from 100K to 10M, with the average value being 500K).

We run each experiment with 8 users (fraud detection experts from company XYZ) and as the variance was less than 2% we present here the average. We also ran our experiments with 10 student volunteers to determine whether the level of expertise affects the results. To simulate the work of a domain expert, we split each dataset into two parts of approximately the same size, before and after a certain point in time. We advanced in time from this point and examined, at different points in time, how the expert adapts the rules in response to transactions arriving up to that point.

We compared the performance of RUDOLF to three alternative solutions, to be detailed below. For each of the algorithms and each of the datasets, we varied the number of new transactions arriving between consecutive rounds of rule refinement. The number of new transactions varies from 10% to 20% of the dataset, with the default being 10%, and this corresponds closely to what happens in real-life between rounds of rule refinement.

Baseline algorithms We consider the two extreme baselines: A *fully-manual* setting, where rules are manually refined by experts without the help of the system (the current setting that is used by company XYZ experts in their daily work), and a *fully-automatic* setting that uses the risk score produced by the ML algorithm and a single rule that selects fraudulent transactions based on their risk scores. Observe that this algorithm essentially generates a single new rule of the form *score greater than threshold* (rather than refining an existing set of rules). We also compared to the baseline algorithm *No Change*, which denotes the given rules without any changes.

Observe that the fully manual setting is arguably our “toughest” competitor since the rules are modified by experts and the experts are not limited by any time constraint to refine the rules.

In addition to the above, we also consider a variant of RUDOLF, denoted RUDOLF⁻, that automatically refines the existing set of rules by accepting the modifications proposed by the system without consulting an expert. We also considered RUDOLF^{-s}, which is the system RUDOLF that does not refine categorical attributes (and hence does not use ontologies) of rules. To the best of our knowledge, all existing systems refine only numerical attributes of rules. Hence the performance of RUDOLF^{-s} will allow us to understand how RUDOLF compares with systems that are only restricted to refine numerical attributes. In fact, we discovered that RUDOLF^{-s} gives almost same results as the fully-manual system and also RUDOLF⁻. Hence, we omit the results of RUDOLF^{-s} completely.

Measurements In our experiments, we measured the efficiency of the algorithms in terms of the effectiveness of the derived rules and the amount of time that the domain experts saved as a consequence of using our system. We also measured the running time required by RUDOLF to select the proposed modifications.

For our datasets this was always at most one second, and we thus omit the exact measures.

To measure the effectiveness of a set of rules derived by any of the methods, we consider its *prediction quality*, namely how correctly it identifies future frauds. For that, we examine the set of transactions from the given point in time where the rules were derived and until the end of the dataset. For these future transactions we count, for each set of rules, the percentage out of all fraudulent (resp. legitimate) transactions that it identifies (resp. wrongly classifies as fraudulent).

To understand of how many modifications each method entailed, we also computed the cumulative number of rule updates that each method required. We measure this only for RUDOLF, RUDOLF⁻, and the fully-manual methods, which directly change existing rules.

Finally, we measured the time the experts took to refine the rules.

Results We first report on our experiments with the domain experts. Our first experiment examines the performance of the algorithms as time advances, with all parameters set to their default value. As explained above, at each point in time, (i.e. after a certain percentage of the transactions has been observed), the algorithms are invoked to derive a corresponding updated set of rules. Figure 3(a) shows the (cumulative) number of modifications that RUDOLF, fully-manual and RUDOLF⁻ method performed to the rules. We see that RUDOLF performs less modifications than its competitors.

We can see this more clearly in Figure 3(b), which illustrates the prediction quality of the derived sets of rules, in terms of the percentage of misclassified future transactions (lower percentage of error implies better prediction quality). RUDOLF performs the best, providing the best prediction. The fully manual rule derivation provides less accurate predictions, though still better than the two automatic competitors. Among the two, RUDOLF⁻, that incrementally refines the rules, still performs better than the threshold-based ML approach.

We note that the difference in performance between RUDOLF⁻ and RUDOLF demonstrates the importance of incorporating experts and their domain knowledge in the loop.

For the experiments above, the rules were periodically refined in hops of 10% of the transactions. For different hops sizes, the results are also similar, except that convergence naturally arrives after fewer (proportionally) iterations for larger hops.

Our next experiment examines the performance of the algorithms for varying dataset, with almost the same percentage of fraud, but different sizes. The size had no significant effect on the number of rule modifications performed by the algorithms, but the prediction quality slightly improved as more data was available. Figure 3(c) illustrate, for varying dataset sizes, the prediction quality of the rules after the first refinement round, in terms of the percentage of misclassified transaction. Here again, lower percentage means better quality. As before, RUDOLF yields best results. We can see that the error of all algorithms slightly decreases as the size of the data set grows. The improvement is only small as fraudulent transactions of the existing fraud patterns are distributed throughout the datasets, so the additional data reveals some, but not huge, amount of new information. Similar results were obtained for the following refinements rounds and we thus omit the graphs.

Next, we examine the performance of the algorithms for varying percentages of fraudulent transactions. We took 4 different

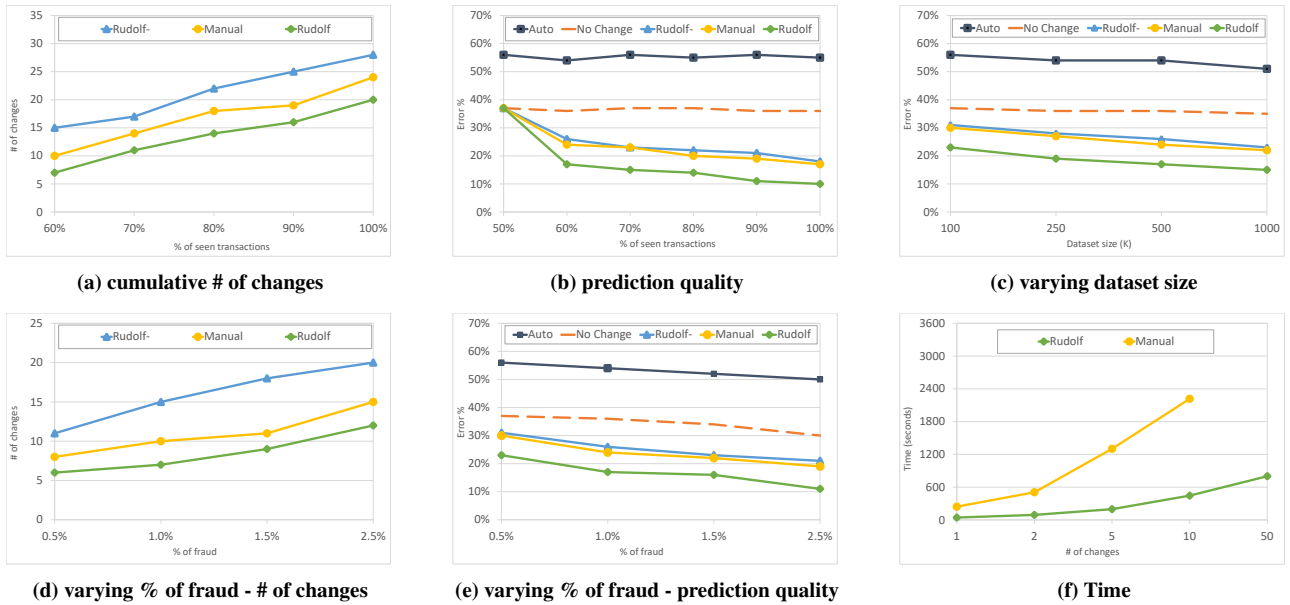


Figure 3: Experimental results

customers databases of roughly the same size, but different fraud percentages (0.5% to 2.5%). All other parameters are set to their default values. Figures 3(d) 3(e) show, respectively, the number of rule updates and percentage of error after the first refinement round. We can see that an increased number of fraudulent transaction entails more rule modifications to capture them. The classification error slightly increases with more fraudulent transactions, but here again RUDOLF achieves the lowest error.

Finally we note that rule refinement with RUDOLF not only consistently yielded superior fraud prediction, but also reduced the time required from the experts by a factor between 4 to 5. (Around 50 seconds per round for RUDOLF compared to 4-5 minutes without). We measured time of our experts performance, depicted in figure 3(f). We asked them to fix up to 50 problematic transactions in both manual and automatic way. Interestingly, no expert finished all 50 fixes in the manual mode (a well-trained expert from company XYZ usually can fix 30-40 transactions per work-day). The fact that RUDOLF leads in performance across all parameters is interesting as the rules derived manually by experts are typically considered as ground truth and yet, RUDOLF is able to do better (i.e., with less changes, and with lower percentage of error) with less data. This was consistent in all the experiments. Furthermore, all users reported that working with RUDOLF was convenient and effective in the sense that the rules/modifications proposed by the system helped them identify and focus on the problematic rules and the needed treatment. To conclude we observed that around 75% of the modifications were condition refinements, 20% rule splits, and 5% rule addition.

Interestingly, our experiments with novice users (student volunteers) show similar trends. In particular, also for novice users, the rules generated with the assistance of RUDOLF were of best quality and produced much faster than in all alternatives. We omit the graphs for space constraints and only note that as expected, compared to the domain experts, the overall prediction quality, even with RUDOLF, was lower (by about 5%) than for the experts, but still significantly better (by 25%) than what the novice users would have achieved alone.

6 RELATED WORK

The identification of fraudulent transactions is essentially a classification problem. Classification has been a fundamental problem in machine learning and data management [14, 15], and crowdsourcing has recently emerged as a major problem solving paradigm [16]. Many classification works have used crowdsourcing to obtain training data for learning [17–20]. This is complimentary to our work: In RUDOLF the crowd (of experts) is employed to maintain classifications rules, which might have been initially learnt through such training. Besides learning-based models, rules are also used for classification. Most of the previous research in rule-based classifiers focus on how to learn rules from the training data. In contrast, [21] employs both learning and analyst experts that manually create classification rules using regular expressions. In [22] that describes LinkedIn’s job title classification system, experts and crowdsourcing are also heavily used. In both cases however the ongoing refinement of rules in a changing environment, which is the focus of RUDOLF, is not considered.

In addition to machine learning-based methods, there are multiple fraud-detection techniques that have been considered. For example [4] uses a decision tree, defined recursively for nodes and edges of the tree and using the ratio between number of transactions that satisfy some condition to label them accordingly. Other methods, e.g. [5], are based on genetic programming, used to classify transactions into suspicious and non-suspicious ones. Another class of the algorithms for fraud detection is based on clustering techniques. An example is [6] that clusters users based on common behavior and then considers as suspicious the transactions that take the user outside its cluster. Bayesian networks are used both to detect fraud in telecommunications (e.g. [23]) and in the credit card industry (e.g. [24]). Neural networks are also used for fraud detection. For instance [25] presents an online fraud detection system, based on a neural classifier. All these techniques are complimentary to ours and can be used to deriving the initial base-set of rules.

Another class of work similar RUDOLF is that of Concept Drifts, which are changes that occur on the distribution of the input that affects the learning system and thus the output. [8] deals with concept drifts by using sliding window that adaptively remembers more or less items from the training set (the closest

past) according to whether it recognized a concept drift or not. Other system ([9]) is classification system based on decision rules. Even though these systems can compute the nearest neighbors for the closest rules and generalization for numerical values, they do not support generalization and specification on categorical attributes, do not involve a human expert in the loop, and do not allow configuration of weight for different kind of errors (false positives and false negatives).

Finally, if we view our transaction relation as the source database and the set of fraudulent transactions as our target database, then the work on deriving queries or schema mappings based on source-target databases (e.g., [26–28]) is also relevant. Similarly, techniques for rule mining and, in particular, inductive logic programming (e.g., [29]) can also be used for fraud detection, where the fraudulent transactions can be seen as positive examples and the legitimate transactions can be seen as negative examples. However, the language of schema mappings and inductive logic programming are different from our rule language and more importantly, the rules derived cannot be interactively adapted.

7 CONCLUSION AND FUTURE WORK

We present RUDOLF, a novel system that assists domain experts in defining and adapting rules in dynamic environments. We show that the problem of identifying the best candidate adaptation for a core language is NP-hard and present PTIME heuristic algorithms for determining the set of rules to adapt and working interactively with the domain experts until they are satisfied with the resulting rules. Our experiments with real-world data sets demonstrate the promise that RUDOLF is an effective and efficient tool for rule refinement.

One direction for future work is the use of more sophisticated cost model. Instead of associating a cost with every modification made to a condition in the rule, one can vary the cost depending on the attribute or even rule that is modified and these costs/weights can be learned or adjusted based on user feedback, satisfaction of the suggested modification etc. Similarly, the parameters α , β and γ used in our cost formula to weight the importance of misclassifying of fraudulent/legitimate/unlabeled transactions may also be dynamically adapted based on such user feedback.

Acknowledgements This work has been partially funded by the European Research Council under the FP7, ERC grant MoDaS, agreement 291071, and by grants from the Blavatnik Cyber Security center and the Israel Innovation Authority. Work was done while Tan was at UCSC. Tan was partially supported by NSF grant IIS-1524382 at UCSC.

REFERENCES

- [1] “The us sees more money lost to credit card fraud than the rest of the world combined,” <http://read.bi/18Gin67>.
- [2] “Card fraud worldwide,” http://nilsonreport.com/publication_chart_and_graphs_archive.php?year=2015.
- [3] “How credit card companies spot fraud before you do,” <http://money.usnews.com/money/personal-finance/articles/2013/07/10/how-credit-card-companies-spot-fraud-before-you-do>.
- [4] A. I. Kokkinaki, “On atypical database transactions: Identification of probable frauds using machine learning for user profiling,” *KDEX*, 1997.
- [5] P. J. Bentley, J. Kim, G.-H. Jung, and J.-U. Choi, “Fuzzy darwinian detection of credit card fraud.”
- [6] R. J. Bolton and D. J. Hand, “Statistical fraud detection: A review,” *Statistical Science*, vol. 2002, pp. 235–255, 2002.
- [7] A. Chapman and H. V. Jagadish, “Why not?” in *SIGMOD*, 2009, pp. 523–534.
- [8] G. Widmer and M. Kubat, “Learning in the presence of concept drift and hidden contexts,” *Machine Learning*, vol. 23, no. 1, pp. 69–101, 1996.
- [9] F. J. Ferrer-Troyano, J. S. Aguilar-Ruiz, and J. C. R. Santos, “Data streams classification by incremental rule learning with parameterized generalization.”
- [10] C. Phua, V. C. S. Lee, K. Smith-Miles, and R. W. Gayler, “A comprehensive survey of data mining-based fraud detection research,” 2010.
- [11] T. Milo, S. Novgorodov, and W. Tan, “Rudolf: Interactive rule refinement system for fraud detection,” *PVLDB*, vol. 9, no. 13, pp. 1465–1468, 2016.
- [12] M. Shindler, A. Wong, and A. Meyerson, “Fast and accurate k-means for large datasets,” in *NIPS*, 2011, pp. 2375–2383.
- [13] “DBPedia,” <http://dbpedia.org>.
- [14] T. M. Mitchell, *Machine learning*, ser. McGraw Hill series in computer science. McGraw-Hill, 1997.
- [15] R. Ramakrishnan and J. Gehrke, *Database management systems (3rd ed.)*. McGraw-Hill, 2003.
- [16] A. Doan, R. Ramakrishnan, and A. Y. Halevy, “Crowdsourcing systems on the world-wide web,” *Commun. ACM*, vol. 54, no. 4, pp. 86–96, 2011.
- [17] S. Vijayanarasimhan and K. Grauman, “Large-scale live active learning: Training object detectors with crawled data and crowds,” in *CVPR*, 2011, pp. 1449–1456.
- [18] V. Ambati, S. Vogel, and J. G. Carbonell, “Active learning and crowd-sourcing for machine translation,” in *LREC 2010*.
- [19] E. Kamar, S. Hacker, and E. Horvitz, “Combining human and machine intelligence in large-scale crowdsourcing,” in *AAMAS*, 2012, pp. 467–474.
- [20] D. R. Karger, S. Oh, and D. Shah, “Iterative learning for reliable crowdsourcing systems,” in *NIPS 2011*.
- [21] C. Sun, N. Rampalli, F. Yang, and A. Doan, “Chimera: Large-scale classification using machine learning, rules, and crowdsourcing,” *PVLDB*, vol. 7, no. 13.
- [22] R. Bekkerman and M. Gavish, “High-precision phrase-based document classification on a modern scale,” in *KDD 2011*, 2011, pp. 231–239.
- [23] K. J. Ezawa and S. W. Norton, “Constructing bayesian networks to predict uncollectible telecommunications accounts,” *Intelligent Systems*, 1996.
- [24] S. Maes, K. Tuyls, B. Vanschoenwinkel, and B. Manderick, “Credit card fraud detection using bayesian and neural networks,” in *NAISO*, 2002.
- [25] D. J.R., G. F., S. C., and C. C.S., “Neural fraud detection in credit card operations,” *IEEE Trans. on Neural Networks*, 1997.
- [26] A. D. Sarma, A. G. Parameswaran, H. Garcia-Molina, and J. Widom, “Synthesizing view definitions from data,” in *ICDT*, 2010.
- [27] Q. T. Tran, C. Y. Chan, and S. Parthasarathy, “Query reverse engineering,” *VLDB J.*, vol. 23, no. 5, pp. 721–746, 2014.
- [28] B. Alexe, B. ten Cate, P. G. Kolaitis, and W. C. Tan, “Designing and refining schema mappings via data examples,” in *SIGMOD*, 2011, pp. 133–144.
- [29] L. Galárraga, C. Teflioudi, K. Hose, and F. M. Suchanek, “Fast rule mining in ontological knowledge bases with AMIE+,” *VLDB J.*, vol. 24, no. 6, 2015.

Privacy Preserving Group Nearest Neighbor Search

Yuncheng Wu*
Renmin University of China
Beijing, China
yunchengwu@ruc.edu.cn

Ke Wang
Simon Fraser University
Burnaby, Canada
wangk@cs.sfu.ca

Zhilin Zhang
Simon Fraser University
Burnaby, Canada
zhilinz@sfu.ca

Weipeng Lin
Simon Fraser University
Burnaby, Canada
weipengl@sfu.ca

Hong Chen[†]
Renmin University of China
Beijing, China
chong@ruc.edu.cn

Cuiping Li
Renmin University of China
Beijing, China
cuipingli@ruc.edu.cn

ABSTRACT

Group k -nearest neighbor (k GNN) search allows a group of n mobile users to jointly retrieve k points from a location-based service provider (LSP) that minimizes the aggregate distance to them. We identify four protection objectives in the privacy preserving k GNN search: (i) every user's location should be protected from LSP; (ii) the group's query and the query answer should be protected from LSP; (iii) LSP's private database information should be protected from users, i.e., the users cannot learn more information beyond the answer they requested; (iv) every user's location should be protected from the other users in the group.

We propose the first approach to meet the four privacy goals in the k GNN query. Our approach provides an accurate query answer and does not rely on heavy pre-computation on LSP like previous works. Our approach considers the most hostile environment that any $n - 1$ users in the query group may collude to infer the location of the remaining user. Though we consider k GNN, the proposed privacy preserving approach can be easily adopted to any group query because it treats the query answering (i.e., k GNN) as a black box. Theoretical and experimental analysis suggest that our approach is highly efficient in both user computation and communication while incurring some reasonable overhead on LSP.

1 INTRODUCTION

The embedding of positioning capabilities (e.g., GPS) in mobile devices facilitates the emergence of location-based services (LBS), which allows the users to publish their location data for retrieving desired information from a database maintained by a location-based service provider (LSP). One typical application is the *group k -nearest neighbor* query (k GNN) proposed in [24], also known as aggregate nearest neighbor query [25] or aggregate similarity search [18, 19, 28]. The k GNN query allows a group of n users to retrieve top- k locations from the LSP's database to minimize some aggregate cost function F over all n users. Figure 1 shows that users u_1, u_2, u_3 ($n = 3$) jointly retrieve the top-2 meeting places $\{p_1, p_2\}$ ($k = 2$), where p_1 has the shortest total distance to the users, and p_2 has the second shortest total distance to the users. By generalizing the classic k -nearest neighbor (k NN) query from a single user in a query to multiple users in a query, the

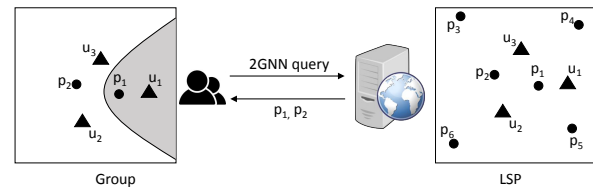


Figure 1: A k GNN query ($k = 2$): $\{u_1, u_2, u_3\}$ retrieves p_1, p_2 as top-2 locations

k GNN query offers richer semantics with broader applications in spatial databases [18, 19, 24, 25, 28].

We consider four privacy concerns that arise in the k GNN query scenario. **Privacy I:** every user's location privacy against LSP since location can reveal the private information of users. **Privacy II:** query privacy and answer privacy against LSP because the query discloses the combination of users' locations and the relationship between users, and the query answer such as a meeting place may disclose the nature of the meeting or event. **Privacy III:** LSP's data privacy against users. LSP's database is the valuable and protected business asset [8, 12, 33] and the principle of least privilege [29] applies where no more information than the requested query answer should be returned to the users. Another reason for this privacy is the pay-per-result model [8, 33] where the users who pay for k results should not receive more than k results. **Privacy IV:** every user's location privacy against other users because users might not trust other users. For example, two business competitors like to query some meeting places but want to hide their own locations from each other. The most hostile environment is the *full user collusion* where $n - 1$ users in the query collude to infer the location of the remaining user. In Figure 1, for example, colluding u_2, u_3 can infer that u_1 is located in the shaded area based on their own locations and the query answer $\{p_1, p_2\}$ received. If the shaded area is too small, u_1 's location privacy may be compromised. **Privacy IV** is required only in the case of $n > 1$.

Although many solutions, such as [1, 3, 12, 13, 17, 21, 26, 27, 30, 34, 36, 37], are proposed for the single user query (i.e., $n = 1$), only a few works addressed the group query (i.e., $n > 1$) [2, 14] but none of them achieves all four privacy concerns. Most existing work achieved **Privacy I** through returning candidate answers (e.g., [3, 13, 14, 17, 21, 26, 30]) or approximate answers (e.g., [1, 2, 34, 37]). However, returning candidate answers not only increases the communication cost but also violates **Privacy III**, while returning approximate answers degrades the answer utility as well as violates **Privacy II** since LSP knows the query answer that users obtained. [12, 27, 36] achieve **Privacy I-III** in

*This work was done when this author visited Simon Fraser University.

[†]Corresponding author.

the single user query case by heavily relying on pre-computing the query answers for all queries. These approaches are not applicable to the group query where the number of possible queries is large. A more detailed discussion of related work is presented in Section 9.

In this paper, we design a privacy preserving approach to the k GNN query for the general case of $n \geq 1$ while protecting **Privacy I-IV**. Our approach has the following novelties. *First*, it eliminates the need for pre-computing all query answers in order to address the privacy issues, while producing the exact answer. Hence, our approach can easily handle a dynamic database on LSP. *Second*, for **Privacy IV** we consider the most hostile environment, called *full user collusion*, where $n - 1$ users may collude to infer the location of the remaining user. *Third*, we aim to reduce user computational cost and communication cost at a reasonable overhead on LSP, which particularly makes sense in the mobile user scenario. *Fourth*, though we consider k GNN, the proposed privacy preserving approach can be easily adopted to any group query because it treats the query answering (i.e., k GNN) as a black box. For example, our approach works with any choice of aggregate cost function F and other location based group queries; to solve the privacy preserving meeting location determination (PPMLD) [5, 16, 31], we can replace the black box for k GNN in our approach with any existing (non-privacy preserving) meeting location determination algorithm.

The rest of the paper is structured as follows.

- Section 2 formulates the privacy preserving k GNN query problem, called PPGNN, where n users jointly retrieve k -nearest neighbors from LSP while meeting the requirements of **Privacy I-IV**.
- Section 3 proposes a solution to PPGNN with $n = 1$. To our knowledge, this is the first work that eliminates the need for pre-computing all the query answers to achieve **Privacy I-III**.
- Section 4 proposes a solution to PPGNN with $n \geq 1$ while achieving **Privacy I-III**.
- Section 5 extends the solution to PPGNN with $n \geq 1$ to achieve **Privacy I-IV** under the full user collusion assumption. As far as we know, this is the first solution that achieves **Privacy I-IV**.
- Section 6 presents an optimization of the PPGNN solution, denoted PPGNN-OPT, to further reduce the communication cost and user computational cost.
- Section 7 theoretically analyzes the performance of the PPGNN solution and PPGNN-OPT solution.
- Section 8 presents empirical results on a real-world dataset, showing that the proposed approaches are highly communication efficient for the privacy guarantees achieved.
- Section 9 reviews the related work.

2 PROBLEM STATEMENT

We formally define the problem studied in this paper. Table 1 summarizes the frequently used notations. We assume that LSP owns a database of Points of Interest (POIs) where each POI has a location (e.g., latitude and longitude) and other associated information, and each user has a location. Both users and LSP are semi-honest: they follow the protocol exactly as specified, but may try to infer others' private information. Users can acquire their locations from satellites anonymously. A base station is responsible for the communication within users and between users and LSP (e.g., mobile communication provider). The base station

Table 1: Summary of notations

Notation	Description
n	the number of users in the group
d	anonymity parameter for Privacy I
δ	anonymity parameter for Privacy II
θ_0	privacy parameter for Privacy IV
$l_{i,*}$	real location of u_i
\mathbb{C}_*	real query $\{l_{1,*}, \dots, l_{n,*}\}$ of the group
\mathbb{L}_i	u_i 's location set
$l_{i,j}$	j -th location in \mathbb{L}_i
\vec{n}, \vec{d}	partition parameters
$[\mathbf{v}]$	encrypted indicator vector for \mathbb{C}_*
\mathbf{A}	answer matrix for all candidate queries
N	the product of two large primes determined by pk

will not collude with LSP, and there is a secure communication channel (e.g., Tor¹) so that LSP cannot infer users' locations by IP addresses. We also assume that LSP does not collude with users. This assumption is reasonable because the penalty of collusion involving LSP is very high, including losing the trust of users and being prosecuted.

2.1 k GNN Query

Let $\mathbb{D} = \{p_1, \dots, p_D\}$ be a database (owned by LSP) of D POIs, and $\mathbb{C}_* = \{l_{1,*}, \dots, l_{n,*}\}$ be the locations of n users. Both \mathbb{D} and \mathbb{C}_* are in a metric space where a spatial distance function dis is defined for any two locations, e.g., Euclidean distance [24, 25], road-network distance [38]. Let F be a monotonically increasing aggregate function,

$$F(p, \mathbb{C}_*) = F(dis(p, l_{1,*}), \dots, dis(p, l_{n,*})) \quad (1)$$

where p can be any POI in \mathbb{D} . Commonly used F includes *sum*, *max* and *min*. For example, with *sum* the query retrieves a meeting place with the minimum total distance to the users, and with *max* the query retrieves a collection place for the troops that leads to the earliest meet time (by minimizing the maximum distance for every troop to reach that place [25]), and with *min* the query retrieves a place that leads to the earliest time for any user to reach that place. In general, the query finds the k best POIs p from \mathbb{D} in an ascending order of $F(p, \mathbb{C}_*)$, as defined below.

Definition 2.1 (kGNN query [28]). Given a spatial database \mathbb{D} , n query locations \mathbb{C}_* , distance function dis , and aggregate function F , a k GNN query ($k \leq D$) retrieves a subset $\mathbb{P} = \{p_1, \dots, p_k\}$ from \mathbb{D} , such that $\forall p_i \in \mathbb{P}$ and $\forall p \in \mathbb{D} - \mathbb{P}$, $F(p_i, \mathbb{C}_*) \leq F(p, \mathbb{C}_*)$, and for $1 \leq i < j \leq k$, $F(p_i, \mathbb{C}_*) \leq F(p_j, \mathbb{C}_*)$. \square

2.2 Privacy Preserving k GNN Query

Definition 2.2 (Privacy preserving kGNN query, or PPGNN). Let \mathbb{D} and \mathbb{C}_* be those in Definition 2.1. A k GNN query is privacy preserving if the following conditions are satisfied:

- (1) **Privacy I:** $\forall i \in [1, n]$, each user u_i 's location $l_{i,*}$ is indistinguishable from d equally likely locations by LSP, $d > 1$;
- (2) **Privacy II:** the query location \mathbb{C}_* and its answer are indistinguishable from δ equally likely candidates by LSP, $\delta \geq d$;
- (3) **Privacy III:** the users can learn nothing more than the requested answer to the query \mathbb{C}_* .
- (4) **Privacy IV:** $\forall i \in [1, n]$, u_i 's location $l_{i,*}$ is hidden in a region from other users, where the size of the region is no less

¹<https://www.torproject.org/>

than θ_0 fraction of the size of the whole location space, $\theta_0 \in (0, 1]$. Under the *full user collusion assumption*, this property should hold even if any $n - 1$ users collude, by sharing their locations, to infer the remaining user's location with the help of the query answer received. \square

Condition (1) guarantees that the probability for LSP to infer a user's location is $\frac{1}{d}$. Condition (2) guarantees that the probability for LSP to infer the group's query, i.e., all users' locations, as well as the query answer, is $\frac{1}{\delta}$, where $\delta \geq d$. Condition (3) guarantees that the users learn only the query answer as defined by the query, in order to protect the LSP's private database. Condition (4) guarantees that any user's location can not be inferred by other users. The strength of these privacy guarantees depends on the setting of the privacy parameters (d, δ, θ_0) , which are specified by the users.

3 SINGLE USER QUERY

In this section, we consider PPGNN with single user, i.e., $n = 1$. In this case, there is no **Privacy IV**, and $\delta = d$. Section 3.1 gives some background knowledge about the generalized Paillier cryptosystem [10]. Section 3.2 presents our solution.

3.1 Generalized Paillier Cryptosystem

Generalized Paillier cryptosystem ε_s ($s \geq 1$) [10] is a probabilistic asymmetric encryption scheme that provides semantic security. Let \mathbb{Z}_{N^s} and $\mathbb{Z}_{N^s}^*$ be the residue class ring module N^s and the prime residue class group module N^s , respectively.

Generalized Paillier cryptosystem is composed of three algorithms (**Gen**, **Enc**, **Dec**): (1) given a security parameter *keysize*, the *key generation* algorithm $(sk, pk) = \mathbf{Gen}(\text{keysize})$ returns secret key sk and public key pk . N is the product of two large primes determined by pk . (2) The *encryption* algorithm $c = \mathbf{Enc}(x, pk)$ maps a plaintext $x \in \mathbb{Z}_{N^s}$ to a ciphertext $c \in \mathbb{Z}_{N^{s+1}}^*$ using pk . (3) The *decryption* algorithm $x = \mathbf{Dec}(c, sk)$ executes reverse operation of encryption by sk , obtaining plaintext x . The exact construction of **Enc** and **Dec**, which can be found in [10], is not important for this work, but the following homomorphism properties of the generalized Paillier cryptosystem are used by our solutions. For simplicity, we omit the public key pk in the **Enc** algorithm. Let x_1, x_2 denote the plaintexts in \mathbb{Z}_{N^s} .

Homomorphic addition \oplus : given two ciphertexts $\mathbf{Enc}(x_1)$, $\mathbf{Enc}(x_2)$, the ciphertext of the sum $x_1 + x_2$ can be obtained by multiplying the ciphertexts, i.e.,

$$\mathbf{Enc}(x_1) \oplus \mathbf{Enc}(x_2) : \mathbf{Enc}(x_1) \cdot \mathbf{Enc}(x_2) = \mathbf{Enc}(x_1 + x_2) \quad (2)$$

Homomorphic multiplication \otimes : given a plaintext x_1 and a ciphertext $\mathbf{Enc}(x_2)$, the ciphertext of the product $x_1 x_2$ can be obtained by raising $\mathbf{Enc}(x_2)$ to the power x_1 :

$$x_1 \otimes \mathbf{Enc}(x_2) : \mathbf{Enc}(x_2)^{x_1} = \mathbf{Enc}(x_1 x_2) \quad (3)$$

Homomorphic dot product \odot : given a ciphertext vector $\mathbf{Enc}(\mathbf{v}) = (\mathbf{Enc}(v_1), \dots, \mathbf{Enc}(v_m))^T$ and a plaintext vector $\mathbf{x} = (x_1, \dots, x_m)$, the ciphertext of the dot product $\mathbf{v} \cdot \mathbf{x}$ can be obtained by:

$$\begin{aligned} \mathbf{x} \odot \mathbf{Enc}(\mathbf{v}) &: (x_1 \otimes \mathbf{Enc}(v_1)) \oplus \dots \oplus (x_m \otimes \mathbf{Enc}(v_m)) \\ &= \mathbf{Enc}(x_1 v_1 + \dots + x_m v_m) \\ &= \mathbf{Enc}(\mathbf{x} \cdot \mathbf{v}) \end{aligned} \quad (4)$$

For simplicity, we write $\mathbf{Enc}(x)$ with ε_1 as $[x]$. One application of Eqn (4) is privately selecting the i -th element in a vector \mathbf{x} without disclosing the value of i . Specifically, let $[\mathbf{v}] =$

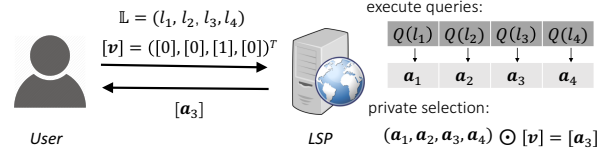


Figure 2: Single user query example (l_3 is real)

$([v_1], \dots, [v_m])^T$ where $v_i = 1$ and $v_j = 0$ for all $j \neq i$. Eqn (4) becomes $\mathbf{x} \odot [\mathbf{v}] = [\mathbf{x} \cdot \mathbf{v}] = [x_i]$, which returns x_i in ciphertext. We will use this private selection to design our solution.

3.2 Proposed Solution

Figure 2 shows the idea of our solution through a running example. The user sends a location set $\{l_1, l_2, l_3, l_4\}$ to LSP, where $d = 4$, as well as an encrypted indicator vector $([0], [0], [1], [0])$ that implicitly specifies that l_3 is the real user location. LSP calculates the query for every location in the location set, producing the answers $\mathbf{a}_1, \mathbf{a}_2, \mathbf{a}_3, \mathbf{a}_4$, and privately selects \mathbf{a}_3 through the result $[\mathbf{a}_3]$ according to the homomorphism properties. This solution has three stages: query generation, query processing, and answer decryption. We discuss each stage in details. We denote the user's real location by l_* .

Query Generation. At first, the user randomly selects $d - 1$ dummy locations from the location space and constructs a location set $\mathbb{L} = \{l_1, \dots, l_d\}$ containing the real location l_* . Then the user creates an indicator vector $\mathbf{v} = (v_1, \dots, v_d)^T$:

$$v_i = \begin{cases} 1, & l_i = l_* \\ 0, & \text{otherwise} \end{cases} \quad (5)$$

After that, the user generates (sk, pk) by calling the key generation algorithm with the security parameter *keysize* (e.g., the most commonly used is 1024 bits [23]), and executes element-wise encryption algorithm on \mathbf{v} , obtaining an encrypted $[\mathbf{v}] = ([v_1], \dots, [v_d])$. Finally, the user sends $\{k, \mathbb{L}, pk, [\mathbf{v}]\}$ to LSP, where k is the number of POIs to retrieve.

Query Processing. After receiving the user's query, LSP computes the k NN query for each location in the location set, resulting in d query answers, $\mathbf{a}_1, \dots, \mathbf{a}_d$, where each query answer is a list of k POIs. The k NN query is computed on *plaintext*, so any plaintext solution for k NN query, such as [24], can be applied. We assume that each query answer \mathbf{a}_i is represented by a vector of integers, $\mathbf{a}_i = (a_{i,1}, \dots, a_{i,m})^T$, such that every element is less than N , where m is the maximum number of integers required for any of the d answers. If the number of integers encoded for a query answer is less than m , 0's are padded as placeholders.

Let $\mathbf{A}^{m \times d} = (\mathbf{a}_1, \dots, \mathbf{a}_d)$ be the query answer matrix. The next theorem suggests that LSP can obtain the encrypted query answer for the real location l_* by a private selection using \mathbf{A} and $[\mathbf{v}]$. The encrypted query answer is returned to the user.

THEOREM 3.1 (PRIVATE SELECTION). *Given an encrypted indicator vector $[\mathbf{v}] = ([v_1], \dots, [v_d])^T$ such that $[v_i] = [1]$ and $[v_j] = [0]$ for all $j \neq i$, and the answer matrix $\mathbf{A}^{m \times d} = (\mathbf{a}_1, \dots, \mathbf{a}_d)$, then*

$[a_i]$ is computed by $A \otimes [v]$ defined as follows:

$$\begin{aligned} A \otimes [v] &= \begin{pmatrix} a_{1,1} & \cdots & a_{d,1} \\ \vdots & \ddots & \vdots \\ a_{1,m} & \cdots & a_{d,m} \end{pmatrix} \otimes \begin{pmatrix} [v_1] \\ \vdots \\ [v_d] \end{pmatrix} \\ &= \begin{pmatrix} [a_{1,1}v_1 + \cdots + a_{d,1}v_d] \\ \vdots \\ [a_{1,m}v_1 + \cdots + a_{d,m}v_d] \end{pmatrix} \\ &= \begin{pmatrix} [0 + \cdots + a_{i,1} + \cdots + 0] \\ \vdots \\ [0 + \cdots + a_{i,m} + \cdots + 0] \end{pmatrix} = \begin{pmatrix} [a_{i,1}] \\ \vdots \\ [a_{i,m}] \end{pmatrix} = [a_i]. \square \end{aligned}$$

The notation \otimes represents the homomorphic matrix multiplication, which executes homomorphic dot product operations in Eqn (4) between each row in A and $[v]$.

Answer Decryption. After receiving $[a_i]$, the user can execute the decryption algorithm on the ciphertext to get the exact query answer a_i for l_* .

In this solution, **Privacy I** is satisfied by anonymizing the user's real location among d locations. **Privacy II** is satisfied because the real query answer is anonymized in d query answers (note $\delta = d$ when $n = 1$) and the selection process is private. Also, LSP returns only the query answer for l_* , so **Privacy III** is satisfied.

4 GROUP QUERY

We now present the PPGNN solution for the group query where $n \geq 1$, which subsumes the solution in Section 3 as a special case. Section 4.1 presents a candidate query generation method that helps satisfy **Privacy II**. Section 4.2 describes the PPGNN solution. Section 4.3 proves the protection of **Privacy I-III**. The protection of **Privacy IV** is addressed in Section 5.

Recall that the real location of each user u_i is denoted as $l_{i,*}$, $1 \leq i \leq n$, and $l_{i,*}$ is hidden in the location set $\mathbb{L}_i = \{l_{i,1}, \dots, l_{i,d}\}$. From \mathbb{L}_i , $1 \leq i \leq n$, LSP can obtain a set of *candidate queries*, where each candidate query is a set of n locations that contains exactly one location from every \mathbb{L}_i . One candidate query is the *real query*, denoted by $\mathbb{C}_* = \{l_{1,*}, \dots, l_{n,*}\}$.

Naive Solution. With $\delta \geq d$, one straightforward method to satisfy both **Privacy I** and **Privacy II** is that every user u_i generates a length δ , instead of length d , location set \mathbb{L}_i , and all users arrange their real locations on the same position in \mathbb{L}_i , $1 \leq i \leq n$. Then LSP can extract one candidate query from the same position in the n location sets, resulting in δ candidate queries. One of these candidate queries is the real query. However, this solution incurs the additional computational cost to generate $\delta - d$ extra dummy locations (e.g., using the dummy generation algorithm [20, 22]) for every user, and the additional communication cost to send the extra dummy locations to LSP. With users' computational power being limited (e.g., mobile devices) and the communication bandwidth being precious, this approach is not practical. To address this special requirement in the mobile user scenario, we propose a solution that aims to reduce the user computational cost and the communication cost at some overhead of the LSP computational cost.

4.1 Candidate Query Generation

Our solution keeps the location set \mathbb{L}_i at the size d but defines a novel protocol for LSP to generate at least δ candidate queries

from the location sets \mathbb{L}_i , $1 \leq i \leq n$. With each \mathbb{L}_i having d elements, d^n candidate queries can be generated by the cartesian product $\times_{i=1}^n \mathbb{L}_i$. We assume $\delta \leq d^n$, otherwise, a larger d should be specified by the users.

Clearly, generating the maximum number of candidate queries, d^n , will satisfy **Privacy II**, but if $\delta \ll d^n$, this means that LSP will compute many unnecessary queries. Our method will generate a minimum number δ' of candidate queries such that $\delta' \geq \delta$, thus, satisfying **Privacy II**. To this end, we partition the user group into α subgroups of the size $\bar{n} = (\bar{n}_1, \dots, \bar{n}_\alpha)$, and partition every location set \mathbb{L}_i into β segments of the size $\bar{d} = (\bar{d}_1, \dots, \bar{d}_\beta)$. $\{\bar{n}, \bar{d}\}$ is called *partition parameters* and is known to both users and LSP. We will determine these parameters shortly. To ensure that the real query will be generated as one of the candidate queries, the following constraint must be satisfied by subgroups and segments: all users arrange their real locations in the *same segment*, and all users from the same subgroup arrange their real locations on the *same position* of that segment. Our query generation in Section 4.2 will enforce this constraint.

Example 4.1. In Figure 4a, the user group is partitioned into 2 subgroups by $\bar{n} = (2, 2)$, and the location sets are partitioned into 2 segments by $\bar{d} = (2, 2)$. All users arrange their real locations in *segment₂*, highlight in red. Also, the users in *subgroup₁* arrange the real locations on the 2-nd position of *segment₂*, and the users in *subgroup₂* arrange the real locations on the 1-st position of *segment₂*. \square

Given the location sets and the partition parameters, LSP generates the candidate queries as follows. Let $G_{i,j,t}$ be the subset of locations from the t -th position of the i -th segment and the j -th subgroup, and let $G_{i,j,:}$ be the subset of locations from the i -th segment and the j -th subgroup. Note that $G_{i,j,:}$ contains \bar{d}_i locations. See Figure 4b for some examples. For $1 \leq i \leq \beta$, LSP computes the candidate queries for the i -th segment by the cartesian product

$$\times_{j=1}^{\alpha} G_{i,j,:} \quad (6)$$

This gives a total number of $\sum_{i=1}^{\beta} (\bar{d}_i)^\alpha$ candidate queries since there are $(\bar{d}_i)^\alpha$ combinations for the i -th segment. Each candidate query is uniquely identified by a sequence of the indexes (i, j, t) for $G_{i,j,t}$, and all candidate queries are listed in the lexicographic order of such indexes. We call this list the *candidate query list*.

Figure 3c illustrates the cartesian product for each segment, generating a total of 8 candidate queries. Figure 3d shows the candidate query list, in which the real query \mathbb{C}_* is at the position 7. This position of the real query can be calculated by the users based on the segment and the positions where the real locations are placed in all location sets. However, LSP does not know this position of the real query because the segment and positions for the real locations are confidentially chosen by the users.

Determining the partition parameters $\{\bar{n}, \bar{d}\}$. The partition parameters are determined by solving the problem

$$\text{minimize}_{\alpha, \beta, \bar{d}} \quad \delta' = \sum_{i=1}^{\beta} (\bar{d}_i)^\alpha \quad (7)$$

$$\text{subject to} \quad \delta' \geq \delta \quad (8)$$

$$\sum_{i=1}^{\beta} \bar{d}_i = d \quad (9)$$

$$\alpha \in \mathbb{N}_{\leq n}, \beta \in \mathbb{N}_{\leq d}, \{\bar{d}_i\}_{i=1}^{\beta} \in \mathbb{N}_{\leq d} \quad (10)$$

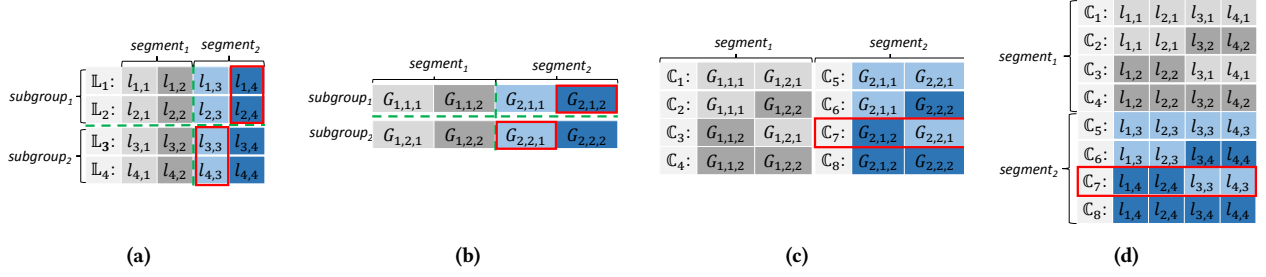


Figure 3: Candidate query generation, where $n = 4$, $d = 4$, and $\delta = 8$: (a) the partition parameters are $\bar{n} = (2, 2)$ and $\bar{d} = (2, 2)$; (b) $G_{2,1,1}$ represents the set $\{l_{1,3}, l_{2,3}\}$ for $segment_2$, $subgroup_1$, and the first position in $segment_2$, while $G_{2,1,*} = \{G_{2,1,1}, G_{2,1,2}\}$; (c) candidate queries $C_1 - C_4$ are generated for $segment_1$, and candidate queries $C_5 - C_8$ are generated for $segment_2$; (d) the candidate query list contains the real query (in red).

δ' in Eqn (7) is the total number of candidate queries generated above. Eqn (9) requires that the sum of all segment sizes should be d , and Eqn (10) requires that the parameters should be integers. The sizes of subgroups are not included in Eqn (7) because they are irrelevant. d, δ, n are constants and the rest are unknown variables. For the single user query, $n = 1$ and $\delta = d$, we can choose $\beta = d$ with each segment size equal to 1. The above is a nonlinear integer programming problem, which is NP-hard [4]. However, the results for frequently used (n, d, δ) can be pre-computed off line (e.g., using open-source solvers [6], Bonmin²). This only needs to be done once.

4.2 PPGNN Solution

The PPGNN solution has three stages: query generation, query processing, and answer decryption. Following [14], we assume that a coordinator user u_c is selected randomly from the query group to assist query generation and answer decryption. Like any other user, no additional trust is assumed of u_c .

Query Generation. Algorithm 1 presents the query generation stage. u_c first determines the partition parameters $\{\bar{n}, \bar{d}\}$ based on $\{n, d, \delta\}$ and calculates the number of candidate queries δ' . As discussed above, the partition parameters could be pre-computed for frequently used $\{n, d, \delta\}$. u_c randomly selects a segment seg from $[1, \beta]$ according to the probability distribution based on the segment sizes, i.e.,

$$\mathcal{P} = \left(\frac{\bar{d}_1}{\bar{d}}, \dots, \frac{\bar{d}_\beta}{\bar{d}} \right) \quad (11)$$

and u_c randomly and uniformly selects a relative position x_j of that segment for the j -th subgroup, and broadcasts pos_j to all users in the j -th subgroup, $1 \leq j \leq \alpha$, where pos_j is the absolute position (over all segments) corresponding to x_j . Then u_c computes the position of the real query C_* in the candidate query list, called *query index*, denoted by QI_{C_*} as follows. Note that the candidate query list is arranged by the lexicographic order of the triples (segment index, subgroup index, position in the segment).

$$QI_{C_*} = \sum_{i=1}^{seg-1} (\bar{d}_i)^\alpha + \sum_{j=1}^\alpha (x_j - 1)(\bar{d}_{seg})^{\alpha-j} + 1 \quad (12)$$

where the first term is the number of candidate queries before reaching the seg -th segment, and the second term is the number of candidate queries before reaching C_* in the seg -th segment.

Example 4.2. In Figure 4a, with $seg = 2$, $\alpha = 2$, and $\bar{d}_1 = 2$, the first term in Eqn (12) is 4, and with $x_1 = 2$ and $x_2 = 1$, the second term is $(2 - 1) * 2^1 + (1 - 1) * 2^0 = 2$, thus $QI_{C_*} = 7$. \square

²<https://neos-server.org/neos/solvers/>

Algorithm 1: Query Generation

Input: $n, d, \delta, k, \text{keysize}, \{l_{i,*}\}_{i=1}^n, \theta_0$;
Output: Query $\{k, pk, \bar{n}, \bar{d}, [\mathbf{v}], \theta_0, \{(i, \mathbb{L}_i)\}_{i=1}^n\}$;

- 1 **Coordinator u_c :**
- 2 $\{\bar{n}, \bar{d}\} \leftarrow$ find the partition parameters; // Eqn (7)-(10);
- 3 $seg \leftarrow$ randomly select a segment by Eqn (11);
- 4 **for** $j \in [1, |\bar{n}|]$ **do**
- 5 $x_j \leftarrow$ randomly and uniformly select from $[1, \bar{d}_{seg}]$;
- 6 $pos_j \leftarrow \sum_{i=1}^{seg-1} \bar{d}_i + x_j$;
- 7 **Send** pos_j to all users in $subgroup_j$;
- 8 $\{sk, pk\} \leftarrow \text{Gen}(\text{keysize})$; // key generation
- 9 $\mathbf{v} \leftarrow$ construct indicator vector by the query index; // Eqn (12)
- 10 $[\mathbf{v}] \leftarrow$ element-wise encryption $\text{Enc}(\mathbf{v}, pk)$;
- 11 **Send** $\{k, pk, \bar{n}, \bar{d}, [\mathbf{v}], \theta_0\}$ to LSP;
- 12 **Every user u_i in $subgroup_j$:**
- 13 receive pos_j from u_c ;
- 14 $\mathbb{L}_i \leftarrow$ generate location set, arranging $l_{i,*}$ on the pos_j -th position;
- 15 **Send** (i, \mathbb{L}_i) to LSP;

Algorithm 2: Query Processing

Input: Query $\{k, pk, \bar{n}, \bar{d}, [\mathbf{v}], \theta_0, \{(i, \mathbb{L}_i)\}_{i=1}^n, \mathbb{D}\}$;
Output: $[a_*]$;

- 1 $\{C_t\}_{t=1}^{\delta'} \leftarrow$ generate the candidate query list by $\{\bar{n}, \bar{d}, \{(i, \mathbb{L}_i)\}_{i=1}^n\}$; // Section 4.1
- 2 **for** $t \in [1, \delta']$ **do**
- 3 $\mathbb{P}_t \leftarrow$ compute k GNN query by (k, C_t, \mathbb{D}) ;
- 4 $\mathbb{P}'_t \leftarrow \text{answerSanitation}(\theta_0, \mathbb{P}_t, C_t)$; // Section 5.2
- 5 $\mathbf{a}_t \leftarrow$ encode \mathbb{P}'_t into integer vector by N stated in pk ;
- 6 $\mathbf{A} = (\mathbf{a}_1, \dots, \mathbf{a}_{\delta'})$;
- 7 $[a_*] = \mathbf{A} \otimes [\mathbf{v}]$; // Theorem 3.1
- 8 **Send** $[a_*]$ to u_c

Then, u_c constructs the encrypted indicator vector $[\mathbf{v}]$ of length δ' , where \mathbf{v} has 1 at the position QI_{C_*} and 0 everywhere else. Finally, u_c sends the query $\{k, pk, \bar{n}, \bar{d}, [\mathbf{v}], \theta_0\}$ to LSP.

Meanwhile, every user $u_i (i \in [1, n])$ in the j -th subgroup arranges its real location $l_{i,*}$ at the received position pos_j in \mathbb{L}_i and dummy locations at the remaining positions, and sends (i, \mathbb{L}_i) to LSP independently. With the user ID i , LSP can reconstruct $subgroup_1$ as the first \bar{n}_1 users, $subgroup_2$ as the next \bar{n}_2 users, and so on. Note that no user, including u_c , knows other users' real locations, because each user sends the location set to LSP directly.

Query Processing. Algorithm 2 presents the query processing stage. After receiving the query from users, LSP generates the candidate query list containing δ' candidate queries as described in Section 4.1, and executes the k GNN query for each candidate query. Line 4 calls the answerSanitation method to ensure that the query answer satisfies **Privacy IV**, which will be presented in Section 5.2. Let $A^{m \times \delta'}$ be the answer matrix for the query answers $(a_1, \dots, a_{\delta'})$. LSP privately selects the query answer $[a_*]$ for \mathbb{C}_* following Theorem 3.1, and sends $[a_*]$ to u_c .

The answer decryption stage is the same as that in Section 3, except that u_c will broadcast the answer a_* to all other users.

4.3 Privacy Guarantees

THEOREM 4.3. *The PPGNN solution satisfies **Privacy I-III**.*

PROOF. Privacy I: The segment seg containing the real locations is selected following the probability distribution corresponding to the segment sizes, Eqn (11), thus, the probability that LSP infers this segment is \bar{d}_{seg}/d . The position for the real locations in the seg -th segment for each subgroup is selected randomly and uniformly, so given the segment seg , the probability that LSP can infer this position is $1/\bar{d}_{seg}$. Overall, the probability for LSP to identify each user's real location is $(\bar{d}_i/d) \cdot (1/\bar{d}_i) = 1/d$.

Privacy II: LSP generates $\delta' (\geq \delta)$ candidate queries and obtain δ' query answers before the private selection. So the probability for LSP to identify group's query \mathbb{C}_* and the query answer is $1/\delta'$, which is no larger than $1/\delta$.

Privacy III: The private selection ensures that only the answer for \mathbb{C}_* is returned, so the users learn no extra POI information beyond the query answer requested. \square

5 FULL USER COLLUSION

So far, the PPGNN solution satisfies only **Privacy I-III**. We now consider answerSanitation on line 4 in Algorithm 2 to enforce **Privacy IV** under the full user collusion assumption. Our first observation is that the only communication within the user group is $\{pos_j\}_{j=1}^n$ that are broadcast from u_c to let all users arrange their real locations in their location sets. This information alone does not allow any $n-1$ colluding users to learn the real location of the remaining user. However, after receiving the query answer, $n-1$ colluding users can infer some possible region that contains the remaining user's real location, with the help of the ranking and location information of the POIs in the query answer. We first present this attack in Section 5.1, and devise a method to prevent this attack and satisfy **Privacy IV** in the rest of the section.

5.1 Inequality Attack

Suppose the users $\{u_1, \dots, u_n\}$ located at the locations $\mathbb{C}_* = \{l_{1,*}, \dots, l_{n,*}\}$ respectively obtain the query answer in the form of k ranked POIs $\mathbb{P} = \{p_1, \dots, p_k\}$ such that

$$F(p_i, \mathbb{C}_*) \leq F(p_j, \mathbb{C}_*), \forall 1 \leq i < j \leq k \quad (13)$$

Without loss of generality, let us assume that u_1 is the attack target and $\{u_i\}_{i=2}^n$ collude together to infer u_1 's location. Therefore, there is only one unknown variable $l_{1,*}$ in Eqn (13) because $l_{2,*}, \dots, l_{n,*}$ as well as the query answer \mathbb{P} are known to the colluding users. Consequently, the colluding users can construct $k-1$ independent inequalities to infer the possible region of $l_{1,*}$:

$$F(p_i, \mathbb{C}_*) \leq F(p_{i+1}, \mathbb{C}_*), \forall 1 \leq i \leq k-1 \quad (14)$$

We refer this inference as the *inequality attack*. Suppose that the area of the solution region for Eqn (14) is θ (in percentage) of

the area of the whole data space, **Privacy IV** is satisfied if and only if $\theta > \theta_0$ for every target user u_1 ; otherwise, i.e., $\theta \leq \theta_0$ for some target user u_1 , **Privacy IV** is not satisfied, and we say the inequality attack succeeds. For instance, **Privacy IV** is not satisfied in Figure 1 if $\theta_0 = 0.5$, because the possible region for u_1 's location (in shaded) is less than half of the whole location space.

5.2 Answer Sanitation

One solution to prevent the inequality attack is that LSP randomly perturbs the order of POIs in \mathbb{P} such that the inequalities in Eqn (14) cannot be correctly constructed. This solution will degrade the utility of the query answer because the users need to know the rank of returned locations. In addition, it is unclear how to ensure that the colluding users cannot reconstruct the original order or a partial order.

Instead, we design a sanitation method for LSP to return the longest prefix $\mathbb{P}' = \{p_1, \dots, p_t\}$ of \mathbb{P} while satisfying **Privacy IV**. In fact, LSP can simulate the inequality attack for every user using a prefix \mathbb{P}' of \mathbb{P} in Eqn (14). If $\theta > \theta_0$ holds on \mathbb{P}' for every target user, where θ is the relative size of the solution region of Eqn (14), \mathbb{P}' is safe of satisfying **Privacy IV**. We will consider how to test $\theta > \theta_0$ shortly. LSP can start with the shortest prefix $\mathbb{P}' = \{p_1\}$, which is always safe, and examine the length t prefix only if the length $t-1$ prefix is safe. The query answer for a candidate query then is the prefix $\mathbb{P}' = \{p_1, \dots, p_t\}$ that is safe, and if $t < k$, the next prefix $\{p_1, \dots, p_{t+1}\}$ is not safe. Note that if the query answer for every candidate query is safe of satisfying **Privacy IV**, the returned answer for \mathbb{C}_* after private selection satisfies **Privacy IV**.

5.3 Testing $\theta > \theta_0$

We now present how to test whether $\theta > \theta_0$ for a target user, where θ is the relative size of the solution region of Eqn (14). One approach is finding the exact solution region of Eqn (14). Unfortunately, finding this solution region is not straightforward, especially for any choice of the aggregation function F and an arbitrary shape of the data space. On the other hand, it is easy to test whether a point $l_{1,*}$ satisfies all inequalities in Eqn (14), that is, falls into the solution region, without explicitly finding the solution region. This observation motivates the following statistic test.

Consider two hypothesis H_0 and H_1 :

$$H_0 : \theta \leq \theta_0, \quad H_1 : \theta > \theta_0 \quad (15)$$

There are two types of errors:

- **Type I Error:** $\Pr(\text{reject } H_0 | H_0 \text{ is true})$ is the probability that a successful attack is not identified for a target user.
- **Type II Error:** $\Pr(\text{not reject } H_0 | H_1 \text{ is true})$ is the probability that a non-attack is identified as a successful attack for a target user.

A small probability for Type I Error provides more confidence on privacy protection and a small probability for Type II Error provides more confidence on better utility of the returned answer. We want to bound these probabilities.

To test whether H_0 should be rejected, LSP can uniformly and independently sample N_H points X_1, \dots, X_{N_H} from the data space. The outcome of each sample is a Bernoulli random variable

$$B = \begin{cases} 1, & \text{if } X_i \text{ satisfies the inequalities in Eqn (14)} \\ 0, & \text{otherwise} \end{cases}$$

The probability of $B = 1$ is equal to the relative size θ of the solution region of Eqn (14). $X = X_1 + \dots + X_{N_H}$ follows a Binomial distribution. For a large N_H , this Binomial distribution can be approximated by the normal distribution. In this case, LSP can reject H_0 through the Z -test statistic [7]: reject H_0 if

$$X > N_H \theta_0 + z_\gamma \sqrt{N_H \theta_0 (1 - \theta_0)} \quad (16)$$

where z_γ is the critical value of the normal distribution and γ is a desired upper bound for the Type I Error probability, i.e., $\Pr(\text{reject } H_0 | H_0 \text{ is true}) \leq \gamma$.

To test if $\theta > \theta_0$ in the answer sanitation, LSP tests the inequality in Eqn (16) instead. From the above discussion, the probability of capturing a successful attack, i.e., $\Pr(\text{not reject } H_0 | H_0 \text{ is true})$, is at least $1 - \gamma$. Since this approach only requires testing if a point satisfies the inequalities in Eqn (14) (for computing X), it is applicable to any choice of the aggregation function F and any shape of the data space.

To determine the sample size N_H , we need to take the probability of Type II Error, i.e., $\Pr(\text{not reject } H_0 | H_1 \text{ is true})$, into consideration. Let θ_1 denote the minimum θ value in H_1 that we want to significantly differentiate from θ_0 . [15] suggests $\frac{\theta_1}{\theta_0} = (1 + \phi)$, where ϕ is the ratio difference between θ_1 and θ_0 . The two types of errors can be bounded given enough samples, as stated below.

THEOREM 5.1 (SAMPLE SIZE [11]). *In the one-tailed hypothesis testing, the sample size N_H required for $\Pr(\text{Type I Error}) \leq \gamma$ and $\Pr(\text{Type II Error}) \leq \eta$ is given by*

$$N_H \geq \left[\frac{z_\gamma \sqrt{\theta_0(1 - \theta_0)} + z_\eta \sqrt{\theta_1(1 - \theta_1)}}{\theta_1 - \theta_0} \right]^2. \quad \square \quad (17)$$

The commonly used γ, η, ϕ are $\gamma = 0.05, \eta = 0.2$, and $\phi = 0.1$. Once the users specify θ_0 , LSP can determine the sample size N_H using Eqn (17).

5.4 Privacy Guarantees

THEOREM 5.2. *The PPGNN solution with the answer sanitation satisfies **Privacy I-IV** under the full user collusion assumption.*

PROOF. Privacy I-III follows from Theorem 4.3. Suppose the returned query answer for \mathbb{C}_* is \mathbb{P}_* , which passes the answer sanitation. Therefore, for every target user u_i in the group query, the Type I Error probability, $\Pr(\text{reject } H_0 | H_0 \text{ is true})$, is bounded by γ . In other words, u_i 's real location is guaranteed to hide in a region that is at least θ_0 (in percentage) of the whole space with the confidence $1 - \gamma$, i.e., **Privacy IV** is satisfied.

A noteworthy point is that the answer sanitation does not affect the protection for **Privacy I-III** because it only reduces the number of POIs in the query answer returned to the users (by returning a prefix of the original top- k answer). The users learn from the reduced list of POIs that the remaining POIs in the answer are not returned because there is an inequality attack. However, without these remaining POIs, the users cannot perform such attacks. \square

6 OPTIMIZED PPGNN

The indicator vector \mathbf{v} with length δ' has only a single 1 (specifying \mathbb{C}_*) and $(\delta' - 1)$ 0's. If δ' is large, additional user computational cost and communication cost are spent on encrypting and transmitting many 0's. In this section, we present an optimization of PPGNN solution, called PPGNN-OPT, to reduce these costs. Let $\llbracket \cdot \rrbracket$ denote the ciphertext generated by the generalized Paillier cryptosystem ε_s with $s = 2$ (see Section 3.1), and as before, let $[\cdot]$

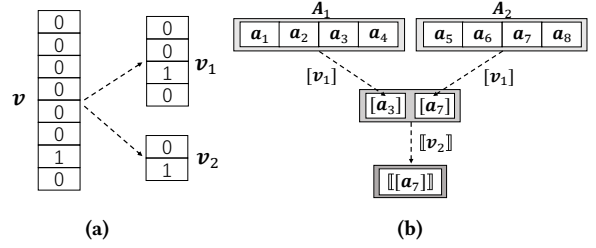


Figure 4: Optimization example: (a) changes on the user side; (b) two phases private selection on the LSP side

denote the ciphertext generated by ε_s with $s = 1$. The encryption and decryption with ε_2 can use the same public key and secret key as those with ε_1 [10].

To explain our optimization, Figure 4 illustrates the changes made on the user side and the LSP side for the example in Figure 3 where $\delta' = \delta = 8$. Instead of using the original indicator vector $\mathbf{v} = (0, 0, 0, 0, 0, 0, 1, 0)^T$ to indicate the position of the real query, u_c uses two small vectors $\mathbf{v}_1 = (0, 0, 1, 0)^T$ and $\mathbf{v}_2 = (0, 1)^T$. On the user side, u_c executes element-wise encryption on \mathbf{v}_1 with ε_1 and executes element-wise encryption on \mathbf{v}_2 with ε_2 , obtaining $\llbracket \mathbf{v}_1 \rrbracket = ([0], [0], [1], [0])^T$, $\llbracket \mathbf{v}_2 \rrbracket = (\llbracket 0 \rrbracket, \llbracket 1 \rrbracket)^T$. These vectors are sent to LSP. In this case, the user computational cost and the communication cost are that for computing and transmitting $\llbracket \mathbf{v}_1 \rrbracket$ and $\llbracket \mathbf{v}_2 \rrbracket$, instead of that for $\llbracket \mathbf{v} \rrbracket$.

On the LSP side, after obtaining the answer matrix A , LSP privately selects the desired answer in two phases. Firstly, LSP partitions A into two sub-matrices (A_1, A_2). For each sub-matrix, LSP executes a private selection using $\llbracket \mathbf{v}_1 \rrbracket$ (Theorem 3.1), resulting in the vector $([a_3], [a_7])$. Secondly, LSP selects the final answer $\llbracket [a_7] \rrbracket$ from $([a_3], [a_7])$ using $\llbracket \mathbf{v}_2 \rrbracket$, by treating the ciphertext of ε_1 as a plaintext of ε_2 in Theorem 3.1. u_c can decrypt $\llbracket [a_7] \rrbracket$ two times to obtain the plaintext query answer a_7 .

In general, if the length of \mathbf{v}_2 is ω , the length of \mathbf{v}_1 is δ'/ω . We assume that δ'/ω is an integer by padding 0's at the end of \mathbf{v} if necessary. We want to choose ω that minimizes the total communication cost between users and LSP. We focus on the ciphertexts transmitted between u_c and LSP because the plaintext location sets transmitted between users and LSP remain unchanged. The length of a ciphertext of ε_2 , which is in \mathbb{Z}_{N^3} , is about twice the length of a ciphertext of ε_1 , which is in \mathbb{Z}_{N^2} . Let L_e denote the length of a ciphertext with ε_1 . We want the integer ω such that

$$\underset{\omega \in \mathbb{N}_{\leq \delta'}}{\text{minimize}} \quad \text{cost}(\omega) = (2\omega + \delta'/\omega + 2m) \cdot L_e \quad (18)$$

where the first term accounts for the length of $\llbracket \mathbf{v}_2 \rrbracket$, the second term accounts for the length of $\llbracket \mathbf{v}_1 \rrbracket$, and the third term accounts for the length of the returned answer (m is the number of ciphertexts required for storing an answer). The optimal ω for Eqn (18) is the nearest integer to $\sqrt{\delta'/2}$, with the minimum communication cost $\text{cost} \approx 2(\sqrt{2\delta'} + m) \cdot L_e$.

In comparison, the communication cost of using the original encrypted indicator vector $\llbracket \mathbf{v} \rrbracket$ is $\text{cost}' = (\delta' + m) \cdot L_e$, where δ' accounts for the length of $\llbracket \mathbf{v} \rrbracket$ and m accounts for the length of the returned answer. The above optimization reduces the cost when $\text{cost} < \text{cost}'$, or $2\sqrt{2\delta'} < \delta' - m$. Since $2\sqrt{2\delta'}$ is positive, $2\sqrt{2\delta'} < \delta' - m$ holds only if $\delta' > m$, and in this case, we have $\delta'^2 + b\delta' + c > 0$, where $b = -(2m + 8)$ and $c = m^2$. The solutions for $\delta'^2 + b\delta' + c > 0$ are $\delta' > r_1$ or $\delta' < r_2$, where

Table 2: Performance analysis

	PPGNN	PPGNN-OPT
Total Communication Cost	$O(nd)L_l + O(\delta')L_e + O(k)L_e$	$O(nd)L_l + O(\sqrt{\delta'})L_e + O(k)L_e$
User Computational Cost	$O(nd)C_l + O(\delta')C_e + O(k)C_e$	$O(nd)C_l + O(\sqrt{\delta'})C_e + O(k)C_e$
LSP Computational Cost	$O(\delta')(C_q + C_s) + O(\delta'k)C_e$	$O(\delta')(C_q + C_s) + O(\delta'k)C_e + O(\sqrt{\delta'k})C_e$

$r_1 = m + 4 + 2\sqrt{2m + 4}$ and $r_2 = m + 4 - 2\sqrt{2m + 4}$. However, since $\delta' > m$ and $r_2 < m$, only $\delta' > r_1$ can be the solution.

In conclusion, on the communication cost, PPGNN-OPT outperforms PPGNN if and only if $\delta' > r_1$ holds. Usually k is not very large and several POIs' information can be encoded into one big integer, therefore m is small and PPGNN-OPT can reduce the cost.

7 PERFORMANCE ANALYSIS

Table 2 summarizes the performance analysis of the PPGNN and PPGNN-OPT solutions in terms of communication cost, user computational cost, and LSP computational cost.

Communication cost. Let L_l and L_e denote the length of a location and the length of a ciphertext of ϵ_1 , respectively. The communication cost of PPGNN includes: n location sets with size d each, i.e., $O(nd)L_l$, $[\mathbf{v}]$ with size δ' , i.e., $O(\delta')L_e$, the returned answer with m encrypted integers that is proportional to the number of POI to be retrieved k , i.e., $O(k)L_e$. The total cost is $O(nd)L_l + O(\delta')L_e + O(k)L_e$. With PPGNN-OPT, the cost for location sets does not change, but the cost for ciphertexts is $O(\sqrt{\delta'})L_e + O(k)L_e$ (see Section 6). Therefore, the total cost is $O(nd)L_l + O(\sqrt{\delta'})L_e + O(k)L_e$.

User computational cost. Let C_l denote the cost for generating a dummy location, and C_e denote the cost for execution on a ciphertext of ϵ_1 (e.g., encryption, decryption). The user computational cost of PPGNN includes: location sets with size d generated by all the users, i.e., $O(nd)C_l$, encryption of $[\mathbf{v}]$ with size δ' computed by u_c , i.e., $O(\delta')C_e$, and decryption of the returned answer, i.e., $O(k)C_e$. The total cost is $O(nd)C_l + O(\delta')C_e + O(k)C_e$. Similar to the analysis of communication cost, for PPGNN-OPT, the total cost is $O(nd)C_l + O(\sqrt{\delta'})C_e + O(k)C_e$.

LSP computational cost. Let C_q denote the cost for executing a k GNN query (e.g., MBM algorithm [24]), and C_s denote the cost for answer sanitation for one candidate query. The LSP computational cost of PPGNN includes: k GNN queries and answer sanitation for $O(\delta')$ candidate queries, i.e., $O(\delta')(C_q + C_s)$, and the private selection on δ' answers with size m , i.e., $O(\delta'k)C_e$. Hence, the total cost is $O(\delta')(C_q + C_s) + O(\delta'k)C_e$. For PPGNN-OPT, the costs for k GNN query and answer sanitation remain unchanged, but the cost for private selection is $O(\delta'k)C_e + O(\sqrt{\delta'k})C_e$ because of the two phases private selection, where the first phase operates on $O(\delta')$ answers and the second phase on $O(\sqrt{\delta'})$ answers. The total cost is $O(\delta')(C_q + C_s) + O(\delta'k)C_e + O(\sqrt{\delta'k})C_e$.

In summary, the communication cost of PPGNN-OPT is asymptotically better than that of PPGNN. However, since the above analysis ignores constant coefficients, in practice, whether PPGNN-OPT is better depends on δ' and m , as discussed at the end of Section 6 (note that $m = xk$, where x is the number of big integers needed to encode one POI). The comparison on user computational cost is similar. The LSP computational cost of PPGNN-OPT is always larger than PPGNN because the second private selection is an extra cost comparing to PPGNN. We will experimentally compare the two solutions in Section 8.

8 EXPERIMENTS

We evaluated the performance of PPGNN (Section 4.2), PPGNN-OPT (Section 6), and the Naive solution (the beginning of Section 4). Since the baseline methods for $n = 1$ and $n > 1$ are different, we consider the single user query scenario ($n = 1$) in Section 8.2, and the group query scenario ($n > 1$) in Section 8.3.

8.1 Experimental Setup

We conducted experiments on a machine with Intel (R) Core (TM) i7-3770 CPU @ 3.40 GHz×8 machine with 15.6G of RAM, running Ubuntu 16.04.1 LTS. All algorithms were implemented in C++. The classic Minimum Bounding Method (MBM) [24] is applied as our plaintext k GNN algorithm in PPGNN, PPGNN-OPT, and the Naive solution, and the aggregation function F is *sum*. We employed the *GMP*³ library for big integer computation and *libhcs*⁴ library for operations of the generalized Paillier scheme. The *keysize* for ϵ_1 is 1024 bits and the *keysize* for ϵ_2 is 2048. Table 3 summarizes all parameters and their ranges.

Dataset. We used a real-world dataset Sequoia⁵, which is widely used in previous studies [12–14, 27, 36]. The dataset contains 62556 POIs from California, including the coordinate and name. As in these previous works, the location space is normalized into a square space, and the real location for every user in a group query was randomly generated as a point in this space. The coordinates of POIs (8 bytes per POI) are returned as the query answer.

Baselines. For $n = 1$, we choose the approximate private k NN query approach, APNN, in [36] as the baseline. In APNN, LSP partitions the data space into grid cells and pre-computes k NN results with respect to the center of each cell and encrypts them. At the query time, the user chooses a square cloak-region containing her location which consists of b^2 cells, and initiates a two-stages cryptographic protocol to retrieve the desired answer. The protocol ensures that LSP does not know which cell the user is located in, nor which answer is retrieved, which ensures **Privacy I-II** with the privacy level b^2 , and the user can only decrypt the requested answer, which ensures **Privacy III**. In our experiments, APNN has a query cloak-region consisting of 5^2 grid cells, which is equivalent to our default setting $d = 25$ for **Privacy I**. Note that APNN produces only approximate answers and relies on pre-computation of k NN results with respect to the center of every cell. This method is not suitable if the exact answer is required or if the database is dynamic. Also, it cannot be extended to the group query scenario because the number of possible queries is significantly large. We did not consider other approaches such as [12, 27] because they are less efficient than APNN according to [36].

For $n > 1$, the first baseline is the incremental pruning private filter (IPPF) algorithm in [14], which is the first work considering users' location privacy in the k GNN query. With the cloak-region technique, IPPF ensures **Privacy I-II** but not **Privacy III-IV**. The

³<http://gmplib.org>

⁴<https://github.com/tiehuis/libhcs>

⁵<http://chorochronos.datastories.org/?q=node/58>

second baseline is the group location privacy (GLP) algorithm in [2], which applies a secure multiparty computation technique for the users to compute their centroid, and LSP returns the k NN query answer with respect to the centroid. GLP ensures **Privacy I** and **III**, but not **Privacy II** and **IV**. A more detailed discussion for IPPF and GLP solutions can be found in Section 9.

Metrics. We measured three dominating costs for the queries: the total *communication cost* (including communications between the user group and LSP, as well as the communications within the user group), the total *user cost* (the sum of all users' computational cost), and the *LSP cost* (all the computations execute on LSP during the query process). In the group query scenario, we also evaluated the number of POIs returned to the users, an indicator of the quality of the answer while resisting the full user collusion inequality attack. We executed 500 queries and reported the average cost.

For the discussion below, the reader is referred to Table 3 for the ranges and default settings of all parameters.

Table 3: Parameters evaluated

	Parameter	Range	Default
$n = 1$	Privacy I parameter (d)	[5, 50]	25
	POI to be retrieved (k)	[2, 32]	8
$n > 1$	Privacy II parameter (δ)	[25, 200]	100
	POI to be retrieved (k)	[2, 32]	8
	user number (n)	[2, 32]	8
	Privacy IV parameter (θ_0)	[0.01, 0.1]	0.05

8.2 Evaluation for Single User Query ($n = 1$)

For $n = 1$, we evaluated PPGNN and PPGNN-OPT by varying d and k . We did not evaluate Naive that is designed for $n > 1$.

Varying d : Figure 5a-5c compares the three costs of PPGNN and PPGNN-OPT for varying d . Note that APNN does not depend on d and is not included. All three costs increase as d increases because each user needs to generate more dummy locations and executes more encryption on the indicator vector, and LSP needs to compute more candidate queries and select the final answer from more query answers. Figure 5a shows that $d = 15$ is the threshold for PPGNN-OPT to outperform PPGNN regarding the communication cost. When $d \geq 15$ (note that $\delta' = d$ when $n = 1$), the communication cost reduction of PPGNN-OPT starts to take effect, which is consistent with the analysis in Section 7. Figure 5b shows a similar trend for the user cost, but with a different threshold $d = 25$ for PPGNN-OPT to beat PPGNN, since the coefficients in Eqn (18) are different with respect to user cost. Usually the cost of operation using ϵ_2 consumes more than 2 times (≈ 3 times in our experiments) than that using ϵ_1 , leading to a larger threshold required. For the LSP cost (Figure 5c), PPGNN always performs better because LSP needs to execute two-phases private selection in PPGNN-OPT. In the mobile user scenario, reducing the communication cost and the user cost has a priority over reducing the LSP cost.

Varying k : Figure 5d-5f reports the performance of PPGNN, PPGNN-OPT, and APNN for varying k . The communication costs (Figure 5d) of the three solutions have a staged growth when k goes up because 15 POIs information can be encoded by a big integer in our settings. Figure 5e shows a similar trend on the user cost for the three solutions. For PPGNN and PPGNN-OPT, the LSP cost (Figure 5f) increases when k becomes larger because the k NN query time and private selection time increase. APNN has

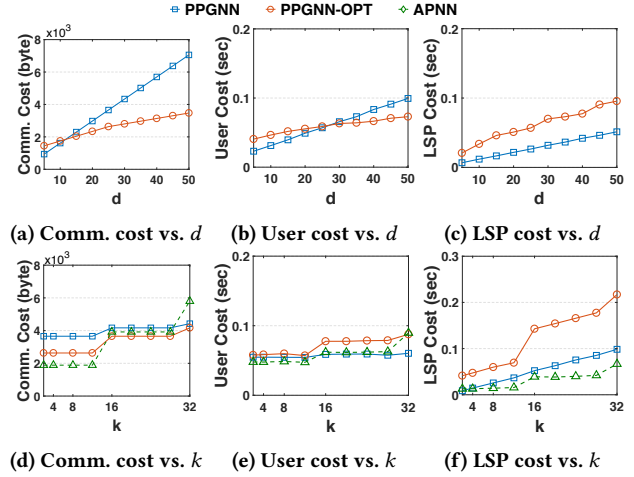


Figure 5: Effect of parameters when $n = 1$

the lowest LSP computational cost because of pre-computed k NN query answers for all grid cells. This computational gain is paid by returning only approximate k NN answers and the potentially expensive update cost for recomputing all k NN answers for a dynamic database.

8.3 Evaluation for Group Query ($n > 1$)

When $n > 1$, we evaluated PPGNN, PPGNN-OPT and Naive in Section 8.3.1 by varying δ , k , n , and θ_0 . We set $d = 25$ by default and did not show its effect because of the relatively stable performance. We experimentally tested for every (n, d, δ) where $n \in [2, 32]$, $d \in [5, 50]$, $\delta \in [50, 200]$ and the average difference between δ' and δ is approximately 1, i.e., $\delta' \approx \delta$. We used the commonly used confidence levels $\gamma = 0.05$, $\eta = 0.2$ and $\phi = 0.1$ in all experiments in the hypothesis testing. The comparison with the baselines IPPF and GLP, which have only two parameters same as ours, k and n , is reported in Section 8.3.2.

8.3.1 Evaluation of Our Approaches. Varying δ : Figure 6a-6c shows the comparison for varying δ . Unlike the comparison for $n = 1$ in Section 8.2, the communication cost and user cost of PPGNN-OPT are much smaller than those of PPGNN and this advantage increases as δ increases. In fact, the size of the encrypted indicator vectors ($[\mathbf{v}_1]$, $[\mathbf{v}_2]$) in PPGNN-OPT is proportional to $O(\sqrt{\delta'})$, whereas the size of the encrypted indicator vector ($[\mathbf{v}]$) in PPGNN is proportional to $O(\delta')$. Therefore, for a large enough δ , although the encryption using ϵ_2 in PPGNN-OPT consumes about three times the cost of the encryption using ϵ_1 , the user cost is still much lower.

The Naive solution incurs the most communication cost because every user in the group needs to send extra $\delta - d$ dummy locations. The LSP costs are almost the same for the three solutions, and are much larger than that for the single user query. Because when $n > 1$, the answer sanitation in Section 5.1 is activated to ensure **Privacy IV**, and this operation dominates the LSP cost. We will discuss more about the LSP cost for answer sanitation in Section 8.3.2.

Varying k : 6d-6f shows the comparison for varying k . The relative comparison of the three solutions is similar to Figure 6a-6c, except that the communication cost and user cost are relatively stable as k increases, with PPGNN-OPT being the best performer. Figure 6f shows that, when k increases, the LSP costs for the

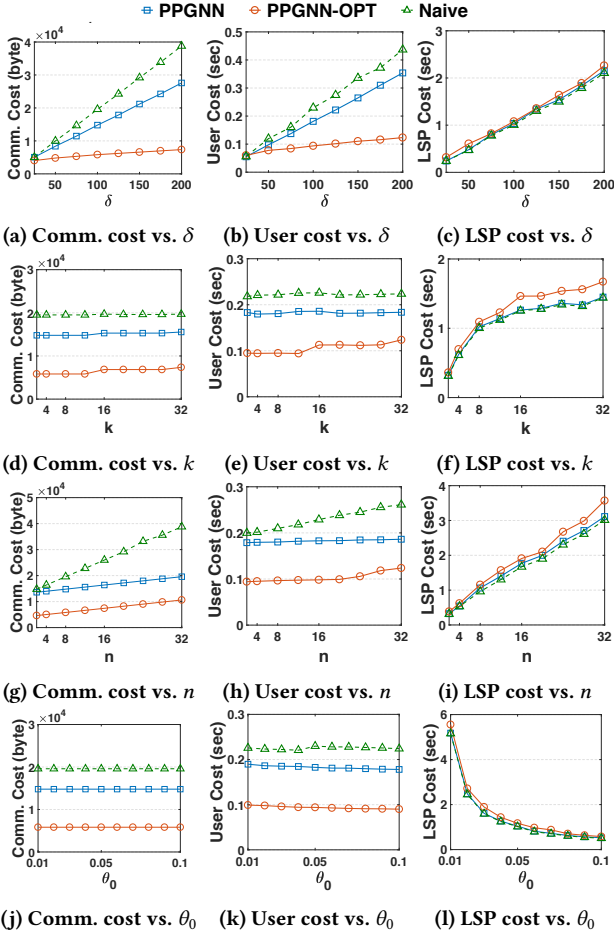


Figure 6: Effect of parameters when $n > 1$

three solutions first go up because the number of inequalities in answer sanitation increases, and become stable when k reaches a number since the number of safe POIs becomes stable as k increases (see Figure 7a).

Varying n : Figure 6g-6i shows the comparison for varying n . The trend is similar to Figure 6a-6c, with PPGNN-OPT being the best performer for the communication cost and user cost. For PPGNN and PPGNN-OPT, n does not affect the size of encrypted indicator vector(s), but for the Naive solution, every user needs to generate and send extra $\delta - d$ dummy locations, which leads to the faster increase in the communication cost and user cost. The LSP cost of three solutions increases linearly because the total number of inequalities considered in the answer sanitation grows linearly with n .

Varying θ_0 : Figure 6j-6l shows the comparison for varying θ_0 . The communication cost and user cost are stable since θ_0 only affects the computation on LSP through the sample size N_H required as in Eqn (17). The LSP cost first decreases greatly and becomes stable as θ_0 increases because the sample size in Eqn (17) behaves this way. In other words, a stronger Privacy IV level means a faster answer sanitation because fewer samples are required in the hypothesis testing.

Number of POIs returned: Recall that the answer sanitation will remove some lower ranked POIs in the top- k answer to ensure Privacy IV under the full user collusion assumption. Thus, the number of POIs actually returned to the users could

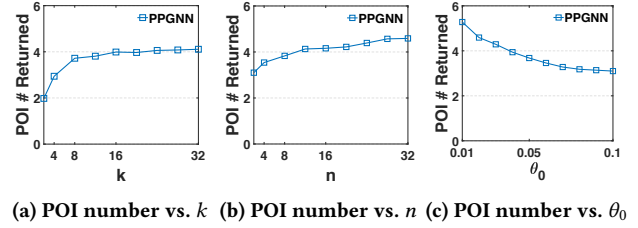


Figure 7: The number of POIs returned per answer

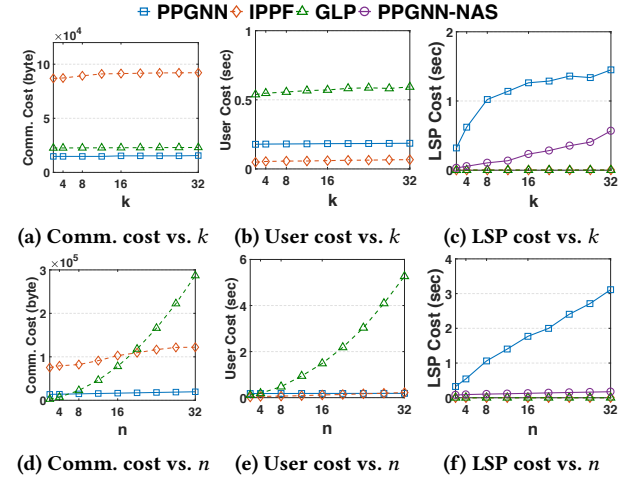


Figure 8: Comparison with other approaches.

be smaller than k . This experiment will study how the answer sanitation affects this number. We only consider PPGNN because PPGNN-OPT and Naive will return the same answer as PPGNN. Note that k , n , and θ_0 can affect this number, but δ cannot.

Figure 7a-7c shows the number of POIs returned per answer for varying k , n , and θ_0 . The default settings of k , n , and θ_0 are 8, 8, and 0.01, respectively. In Figure 7a, as k increases, the number of POIs returned first increases and then becomes stable around 4. This is because, with $n = 8$ and $\theta_0 = 0.01$, 4 inequalities usually lead to a successful inequality attack and a larger k has no additional impact. In Figure 7b, the number of returned POIs rises slightly as n increases. At first glance, this seems counter-intuitive because ensuring no attack on more users is more restrictive. At a closer look, with more users involved in a query, the target user's location $l_{1,*}$ weights less in determining if the inequalities in Eqn (14) hold or not, therefore, there are more choices for $l_{1,*}$, thus, a larger solution region for $l_{1,*}$ and it is easier to add the next POI in the answer. The trend in Figure 7c is expected because a larger θ_0 leads to a stronger Privacy IV, consequently, fewer POIs can be returned.

To conclude, the top 2 to 5 POIs are still returned for a query even after the answer sanitation. In practice, such numbers are usually sufficient because the users might only need to select one from them.

8.3.2 Comparison with Baseline Approaches. We compared PPGNN with the baselines IPPF and GLP for varying k and n . d , δ , and θ_0 specific to PPGNN are set to their default values. We only consider PPGNN because the experiments above have shown that PPGNN-OPT is better than PPGNN. While PPGNN satisfies Privacy IV assuming full user collusion, let PPGNN-NAS denote

Table 4: Comparison with existing work

Group Size	Approaches	Technique	Privacy I	Privacy II	Privacy III	Privacy IV
$n = 1$	[3, 9, 21]	Cloak-Region	✓	✓	×	-
	[17, 30]	Dummy	✓	✓	×	-
	[13, 26]	Private Information Retrieval	✓	✓	×	-
	[1, 34, 37]	Perturbation	✓	×	✓	-
	[12, 27, 36]	Hybrid Techniques	✓	✓	✓	-
	Our approach	Dummy+Paillier	✓	✓	✓	-
$n > 1$	[14]	Cloak-Region	✓	✓	×	×
	[2]	Secure Multiparty Computation	✓	×	✓	×
	Our approach	Dummy+Paillier	✓	✓	✓	✓

the relaxed PPGNN that satisfies **Privacy IV** assuming no user collusion. So PPGNN-NAS does not run the answer sanitation. For IPPF, we chose the query rectangle area (for each user) to be 0.0005% of the data space, which is comparable to choosing $d = 25$ locations (our default setting) as the location set from 5000,000 addresses in California⁶. For GLP, we chose the same *keysize* as PPGNN.

Varying k : Figure 8a-8c shows the comparison for varying k . The communication cost of IPPF is much larger than PPGNN and GLP. In fact, IPPF returns all the candidate POIs, which can be several thousands per query on average, to the users, and the users have to filter the candidate POIs. GLP consumes more user cost than PPGNN and IPPF because there are $O(n^2)$ cryptographic operations and every user has to transmit encrypted values to all other users. In Figure 8c, the gap between PPGNN and PPGNN-NAS is the LSP time spent on the answer sanitation, which dominates the LSP cost. IPPF and GLP consume less LSP cost, however, IPPF cannot satisfy **Privacy III** and **IV**, and GLP cannot satisfy **Privacy II** and **IV** and provide only an approximate answer.

Varying n : Figure 8d-8f shows the comparison for varying n . PPGNN is significantly communication efficient than IPPF and GLP. The communication cost for GLP increases quickly with n because the number of transmitted encrypted values is $O(n^2)$. There is a similar trend on the user cost. Again, Figure 8f shows that PPGNN spent significant LSP time on the answer sanitation to deal with the full user collusion, whereas PPGNN-NAS has almost the same LSP time as IPPF and GLP.

8.4 Summary

For $n = 1$, our solutions, PPGNN and PPGNN-OPT, are comparable with or better than the existing solution APNN that heavily relies on pre-computing the answers for all possible queries to reduce the run-time cost. However, APNN produces only approximate answers and the pre-computation means an expensive update cost. For $n > 1$, which is the main focus of this paper, PPGNN and PPGNN-OPT, have significantly smaller communication cost and user cost than existing solutions IPPF and GLP while providing stronger privacy guarantees, i.e., **Privacy I-IV**, and PPGNN-OPT performs better than PPGNN in most cases. We believe that reducing the communication cost and user communication cost is the priority in the mobile user scenario as considered here. The pay for achieving the stronger privacy guarantee is some increase in the LSP cost, especially when dealing with the full user collusion assumption for **Privacy IV**. To our knowledge, this is the first work considering this assumption.

⁶<https://openaddresses.io/>

9 RELATED WORK

Most existing work focus on the single user query case, i.e., $n = 1$. To protect **Privacy I**, some techniques obfuscate user’s exact location in a cloak-region (CR) [3, 9, 21] or use dummy query locations [17, 30]. Another technique [13, 26] is based on private information retrieval (PIR), allowing the user to retrieve a particular record from LSP without revealing which record is retrieving. In these techniques, LSP needs to return a super-set of the exact query answer, which not only increases the communication cost but also sacrifices **Privacy III**. Data transformation [37] and differential privacy approach [1, 34] perturb user’s exact location to a false location, so the query answer is approximate, and meanwhile, **Privacy II** is sacrificed because LSP knows the query answer. The approaches [12, 27, 36] that based on a hybrid of PIR and cryptography technique can protect **Privacy I-III**, but LSP needs to pre-compute the answers to all possible queries. This technique does not work for $n > 1$ because of too many group queries. Also, if a POI information is updated, LSP needs to re-compute all answers, which is too expensive.

For the group query scenario, i.e., $n > 1$, Hashem *et al.* [14] obfuscates each user’s exact location into a region, and LSP executes the k GNN query *w.r.t.* these regions, returning a super-set of the query answer. Beside the extra work of filtering the answer set by the users, this approach sacrifices **Privacy III** since extra POI information is returned to the users. **Privacy IV** is also violated because a user’s exact location is compromised if its predecessor and successor collude in the filtering phase [2]. In the work by Talouki *et al.* [2], users compute their centroid by secure multiparty computation (SMC) [35] and LSP returns the k NN answer for that centroid. This approach cannot protect **Privacy II** and **Privacy IV** because LSP knows the query answer and $n - 1$ users can recover the last user’s exact location by their own locations and the centroid.

Table 4 summarizes and compares the above works with our work.

In the privacy preserving meeting location determination (PPMLD) [5, 16, 31], a group of users each chooses a preferred meeting location and send the encrypted locations to the server, who then selects the one to minimize the aggregate distance to all preferred locations. In our work, the query answer is selected from the POI database of LSP and the users specify their current locations, instead of preferred meeting locations. The PPMLD method cannot be adopted to our privacy preserving k GNN problem because their cryptographic selection is specific to PPMLD. On the other hand, our approach can be adopted to PPMLD by replacing the k GNN building block with any existing non-privacy preserving meeting location determination solution.

The data ownership and privacy implication in our problem are different from most works on secure query processing in the outsourcing database (ODB) model [32]. In ODB, the users and the data owner are trusted and the server is not trusted. The data owner outsources the encrypted database to the server and the user retrieves the query answer from the server. The privacy objective is to prevent the server from learning anything about the database and the user query. In our problem, LSP owns the data and multiple users jointly specify a query, and no party trusts anyone except herself.

10 CONCLUSION

This work identifies four privacy objectives for the group k -nearest neighbor query, k GNN, and designs a privacy preserving k GNN solution, PPGNN. To our knowledge, this is the first work that address all of these privacy objectives. Though we consider k GNN, the proposed privacy preserving approach can be easily adopted to any group query because it treats the query answering (i.e., k GNN) as a black box.

ACKNOWLEDGMENT

This work is supported by National Science Foundation of China (No.61532021, 61772537), National High Technology Research and Development Program of China (863) (No.2014AA015204), National Key R & D program of China (No.2016YFB1000702), National Basic Research Program of China (973) (No.2014CB340403). Ke Wang's work was partially supported by a discovery grant from The Natural Sciences and Engineering Research Council of Canada (NSERC). This work was partially done when some authors worked in SA Center for Big Data Research in Renmin University of China. The SA Center is funded by Chinese National 111 Project Attracting International Talents in Data Engineering and Knowledge Engineering Research.

REFERENCES

- [1] Miguel E. Andrés, Nicolás Emilio Bordenabe, Konstantinos Chatzikokolakis, and Catuscia Palamidessi. 2013. Geo-indistinguishability: differential privacy for location-based systems. In *CCS*. 901–914.
- [2] Maede Ashouri-Talouki, Ahmad Baraani-Dastjerdi, and Ali Aydin Selçuk. 2012. GLP: A cryptographic approach for group location privacy. *Computer Communications* 35, 12 (2012), 1527–1533.
- [3] Bhuvan Bamba, Ling Liu, Péter Pesti, and Ting Wang. 2008. Supporting anonymous location queries in mobile environments with privacygrid. In *WWW*. 237–246.
- [4] Pietro Belotti, Christian Kirches, Sven Leyffer, Jeff Linderoth, James Luedtke, and Ashutosh Mahajan. 2013. Mixed-integer nonlinear optimization. *Acta Numerica* 22, 1–131.
- [5] Igor Bilogrevic, Murtuza Jadliwala, Vishal Joneja, Kübra Kalkan, Jean-Pierre Hubaux, and Imad Aad. 2014. Privacy-Preserving Optimal Meeting Location Determination on Mobile Devices. *IEEE Trans. Information Forensics and Security* 9, 7 (2014), 1141–1156.
- [6] Pierre Bonami, Mustafa Kilinç, and Jeff Linderoth. 2012. *Algorithms and Software for Convex Mixed Integer Nonlinear Programs*. Springer New York, 1–39.
- [7] Sprinthall R. C. *Basic Statistical Analysis (9th ed.)*. Pearson Education.
- [8] Sunoh Choi, Gabriel Ghinita, Hyo-Sang Lim, and Elisa Bertino. 2014. Secure k NN Query Processing in Untrusted Cloud Environments. *TKDE* 26, 11 (2014), 2818–2831.
- [9] Chi-Yin Chow, Mohamed F. Mokbel, and Xuan Liu. 2006. A peer-to-peer spatial cloaking algorithm for anonymous location-based service. In *ACM-GIS*. 171–178.
- [10] Ivan Damgård and Mads Jurik. 2001. A Generalisation, a Simplification and Some Applications of Paillier's Probabilistic Public-Key System. In *Public Key Cryptography*. 119–136.
- [11] Joseph L Fleiss, Bruce Levin, and Myunghee Cho Paik. 2003. *Statistical methods for rates and proportions; 3rd ed.* Wiley, Hoboken, NJ.
- [12] Gabriel Ghinita, Panos Kalnis, Murat Kantarcioglu, and Elisa Bertino. 2011. Approximate and exact hybrid algorithms for private nearest-neighbor queries with database protection. *Geoinformatica* 15, 4 (2011), 699–726.
- [13] Gabriel Ghinita, Panos Kalnis, Ali Khoshgozaran, Cyrus Shahabi, and Kian-Lee Tan. 2008. Private queries in location based services: anonymizers are not necessary. In *SIGMOD*. 121–132.
- [14] Tanzima Hashem, Lars Kulik, and Rui Zhang. 2010. Privacy preserving group nearest neighbor queries. In *EDBT*. 489–500.
- [15] D.C. Howell. 2013. *Statistical Methods for Psychology*. Wadsworth Cengage Learning.
- [16] Yan Huang and Roopa Vishwanathan. 2010. Privacy Preserving Group Nearest Neighbour Queries in Location-Based Services Using Cryptographic Techniques. In *GLOBECOM*. 1–5.
- [17] Hidetoshi Kido, Yutaka Yanagisawa, and Tetsuji Satoh. 2005. An anonymous communication technique using dummies for location-based services. In *ICPS*. 88–97.
- [18] Feifei Li, Ke Yi, Yufei Tao, Bin Yao, Yang Li, Dong Xie, and Min Wang. 2016. Exact and approximate flexible aggregate similarity search. *VLDB J.* 25, 3 (2016), 317–338.
- [19] Yang Li, Feifei Li, Ke Yi, Bin Yao, and Min Wang. 2011. Flexible aggregate similarity search. In *SIGMOD*. 1009–1020.
- [20] Hua Lu, Christian S. Jensen, and Man Lung Yiu. 2008. PAD: privacy-area aware, dummy-based location privacy in mobile services. In *Seventh ACM International Workshop on Data Engineering for Wireless and Mobile Access*. 16–23.
- [21] Mohamed F. Mokbel, Chi-Yin Chow, and Walid G. Aref. 2006. The New Casper: Query Processing for Location Services without Compromising Privacy. In *PVLDB*. 763–774.
- [22] Ben Niu, Qinghua Li, Xiaoyan Zhu, Guohong Cao, and Hui Li. 2014. Achieving k -anonymity in privacy-aware location-based services. In *INFOCOM*. 754–762.
- [23] Pascal Paillier. 1999. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *EUROCRYPT*. 223–238.
- [24] Dimitris Papadias, Qionghao Shen, Yufei Tao, and Kyriakos Mouratidis. 2004. Group Nearest Neighbor Queries. In *ICDE*. 301–312.
- [25] Dimitris Papadias, Yufei Tao, Kyriakos Mouratidis, and Chun Kit Hui. 2005. Aggregate nearest neighbor queries in spatial databases. *ACM Trans. Database Syst.* 30, 2 (2005), 529–576.
- [26] Stavros Papadopoulos, Spiridon Bakiras, and Dimitris Papadias. 2010. Nearest Neighbor Search with Strong Location Privacy. *PVLDB* 3, 1 (2010), 619–629.
- [27] Russell Paulet, Md. Golam Kaosar, Xun Yi, and Elisa Bertino. 2014. Privacy-Preserving and Content-Protecting Location Based Queries. *TKDE* 26, 5 (2014), 1200–1210.
- [28] Humberto Luiz Razente, Maria Camila Nardini Barioni, Agma J. M. Traina, Christos Faloutsos, and Caetano Traina Jr. 2008. A novel optimization approach to efficiently process aggregate similarity queries in metric access methods. In *CIKM*. 193–202.
- [29] Jerome H. Saltzer and Michael D. Schroeder. 1975. The protection of information in computer systems. *Proc. IEEE* 63, 9 (1975), 1278–1308.
- [30] Pravin Shankar, Vinod Ganapathy, and Liviu Iftode. 2009. Privately querying location-based services with SybilQuery. In *UbiComp*. 31–40.
- [31] Xiaofen Wang, Yi Mu, and Rongmao Chen. 2016. One-Round Privacy-Preserving Meeting Location Determination for Smartphone Applications. *IEEE Trans. Information Forensics and Security* 11, 8 (2016), 1712–1721.
- [32] Wai Kit Wong, David Wai-Lok Cheung, Ben Kao, and Nikos Mamoulis. 2009. Secure k NN computation on encrypted databases. In *SIGMOD*. 139–152.
- [33] David J. Wu, Joe Zimmerman, Jérémy Planul, and John C. Mitchell. 2016. Privacy-Preserving Shortest Path Computation. In *NDSS*.
- [34] Yonghui Xiao and Li Xiong. 2015. Protecting Locations with Differential Privacy under Temporal Correlations. In *CCS*. 1298–1309.
- [35] Andrew Chi-Chih Yao. 1982. Protocols for Secure Computations (Extended Abstract). In *23rd Annual Symposium on Foundations of Computer Science*. 160–164.
- [36] Xun Yi, Russell Paulet, Elisa Bertino, and Vijay Varadharajan. 2016. Practical Approximate k Nearest Neighbor Queries with Location and Query Privacy. *TKDE* 28, 6 (2016), 1546–1559.
- [37] Man Lung Yiu, Christian S. Jensen, Xuegang Huang, and Hua Lu. 2008. SpaceTwist: Managing the Trade-Offs Among Location Privacy, Query Performance, and Query Accuracy in Mobile Services. In *ICDE*. 366–375.
- [38] Man Lung Yiu, Nikos Mamoulis, and Dimitris Papadias. 2005. Aggregate Nearest Neighbor Queries in Road Networks. *TKDE* 17, 6 (2005), 820–833.

Pattern Search in Temporal Social Networks

Maximilian Franzke
 Institute of Informatics,
 Ludwig-Maximilians-Universität
 Munich, Germany
 franzke@dbs.ifi.lmu.de

Tobias Emrich
 Institute of Informatics,
 Ludwig-Maximilians-Universität
 Munich, Germany
 emrich@dbs.ifi.lmu.de

Andreas Züfle
 Department for Geography and GeoInformation Science
 George Mason University
 Fairfax, Virginia, United States
 azufle@gmu.edu

Matthias Renz
 Department for Computational and Data Sciences
 George Mason University
 Fairfax, Virginia, United States
 mrenz@gmu.edu

ABSTRACT

Due to the wide availability of social media and the wide range of real-life and human-centered applications, social networks have become an attractive research area. However, the temporal aspect of relations between entities in a social network has been widely ignored. We argue that the temporal aspect of social networks is the key to understand interactions and other phenomena happening in these networks and should thus be considered more closely. In this work we address the problem of pattern search in temporal social networks, thus finding all occurrences of a temporal pattern in a large temporal social network. As a first step, we define a temporal pruning criterion, which allows to quickly reduce the search space of candidates. Then, we present an index structure which allows to quickly find the occurrences of simple temporal network structures, from which more complex query structures can be derived from. Our experimental evaluation on a real-world temporal social network shows the effectiveness of our pruning approach and our proposed index structures, reducing the search-time for small temporal patterns by many orders of magnitude.

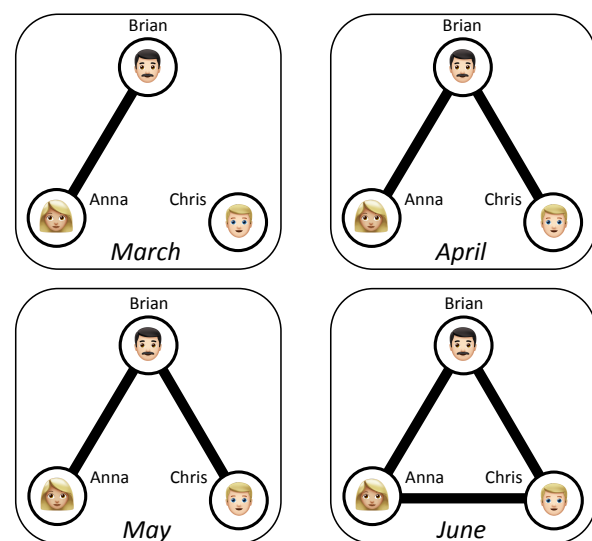


Figure 1: Example evolution of a social network.

1 INTRODUCTION

Social networks and other interaction networks¹ are dynamic by nature. Bonds of friendship can last for an eternity, but often break and fade away over time. Active collaborations between researchers naturally change over time. Thus, any social network today will look significantly different tomorrow. Keeping in mind the dynamic nature of human beings, the history of a social network showing how interactions between individuals evolve over time should be considered when inferring knowledge from it, because the knowledge about the evolution of a social network yields further semantic information. For instance, in a collaboration network it might be interesting to see in which order a

group of researchers formed a network, which researchers only have short-term collaborations and which have rather long and sustainable research collaborations. However, existing solutions that consider only a snapshot in time, or building the union of all past collaborations, are not able to take such information into account. By considering this dynamic aspect of a social network, it becomes possible to identify more interesting and meaningful patterns. As a minimal example, consider Figure 1, showing an evolving social network. Initially, only one social link exists, between Anna and Brian. In the next month, an additional link is added between Brian and Chris. Finally, the triangle is closed with a link between Chris and Anna. In this example, Brian might act as a social hub, who brings Anna and Chris together. In practice, more detailed temporal social patterns could matter, as for instance the duration of a social link between Anna and Chris, which was induced by Brian, may be of interest. The temporal development of edges like in our example has been addressed for more than a hundred years through the concept of ‘triadic closure’, where the future creation of the third connection in a

¹Though this paper mainly refers to social network, the proposed concepts are also applicable for other interaction networks, e.g. economical networks, etc.

triangle is tried to be estimated, e.g. through link prediction in social networks. A more complex temporal structure in a social graph is the ‘microtaboo’, where it is frowned upon when persons *Alice* and *Dave* want to engage in a relationship, but there exist prior relationships between *Alice* and *Bob*, *Bob* and *Carol*, and *Carol* and *Dave* (“don’t date your ex-girlfriend’s boyfriend’s ex-girlfriend”) [2].

Another example are communication and transportation networks, where links between nodes and hubs are only established temporarily. Routing policies of computer networks may decide differently at various time points on how to create links between network nodes in order to transmit data. Transportation networks with feeder trucks, cargo aircraft and delivery vehicles link hubs differently according to current demand. To analyze the behaviour of those networks, a temporal pattern analysis of the interaction graph can help in mining information from the graph’s history to deduct findings and information for future optimization.

In this paper we address the problem of efficient evolution pattern search in large temporal social networks. Our approach bridges the gap between social network analysis and temporal logic. The contributions of this paper can be summarized as follows:

- We formally introduce the temporal subgraph matching query as a new problem.
- We introduce a language to express such pattern-based queries.
- We introduce and discuss several query filter strategies.
- We propose an index structure that allows us to search for temporal subgraph patterns in large temporal social network graphs.
- We provide a broad evaluation of the performance of our approaches based on real world datasets and show that our approach significantly outperforms state-of-the-art approaches. Though our problem is a generalization of the subgraph isomorphism problem, which is known to be NP-complete [6, 11], we can show that our index-based solution is able to find simple temporal social patterns in large real-world social networks efficiently.

We define the problem of temporal social subgraph search in Section 2. In Section 4, we propose filter strategies and introduce our new indexing method in Section 5. Our experimental evaluation is given in Section 7.

2 PROBLEM DEFINITION

We first introduce our representation of a temporal social network which we define as a graph where each node refers to an individual and each edge between two nodes is associated with a discrete function that maps time to the domain $\{0,1\}$ specifying the presence and absence of the edge over time.

Definition 2.1 (Temporal Social Graph). Let $\mathcal{T} = \{0, 1, \dots, m\}$ be a discrete time domain. A temporal social graph (TSG) $G = (V^G, E^G, F^G)$ is a graph, where $V^G = \{v_1^G, v_2^G, \dots, v_n^G\}$ is the set of nodes, $E^G \subseteq V^G \times V^G$ the set of links (with $(v_i^G, v_k^G) \in E^G$), and $F^G = \{f_{v_i^G, v_k^G}^G(t) | (v_i^G, v_k^G) \in E^G\}$ a set of discrete time-dependent functions, where $f_{v_i^G, v_k^G}^G(t) \in \{0, 1\}$ describes the existence of a

connection between v_i^G and v_k^G (0 indicating no connection and 1 indicating there is a connection) at time $t \in \mathcal{T}$. Furthermore, an edge (v_i^G, v_k^G) is only an element of E^G if $\exists t \in \mathcal{T} : f_{v_i^G, v_k^G}^G(t) \neq 0$.

Based on this definition we are now able to define temporal subgraph matching which finds a subgraph from a TSG that exactly matches a given temporal subgraph query.

Definition 2.2. [Temporal Subgraph Matching] Let $G = (V^G, E^G, F^G)$ be a TSG defined on the time domain $\mathcal{T} = \{0, 1, \dots, m\}$. And let $q = (V^q, E^q, F^q)$ be a query TSG defined on the time domain $\mathcal{T}^q = \{0, 1, \dots, t^q\}$ where $t^q \leq m$. A temporal subgraph matching query retrieves the set \mathcal{S} of all temporal subgraphs $S \ni S = (V^S \subseteq V^G, E^S \subseteq E^G, F^S \subseteq F^G)$, such that there exists a bijection $h : V^q \rightarrow V^S$ and $\Delta_t \in \{0, \dots, m - t^q\}$ that satisfies $\forall (v_i^q, v_k^q) \in E^q : (h(v_i^q), h(v_k^q)) \in E^S$ and $\forall t \in [0, t^q] : f_{v_i^q, v_k^q}^q(t) = f_{h(v_i^q), h(v_k^q)}^S(t + \Delta_t)$.

An example for a temporal social graph G is given in Figure 2(a). For convenience we labeled the edges (v_i, v_k) with the set of time steps $t \in \mathcal{T}$ for which the function $f_{v_i, v_k}(t) = 1$. Figure 2(b) illustrates a temporal subgraph query q . A pattern match of q can be found at \clubsuit in G for $\Delta_t = 2$, $h(v_1^q) = v_4^G$, $h(v_2^q) = v_2^G$ and $h(v_3^q) = v_1^G$. A more sophisticated query q' is depicted in Figure 6, which matches for $h(v_1^{q'}) = v_{11}^G$, $h(v_2^{q'}) = v_8^G$, $h(v_3^{q'}) = v_9^G$, $h(v_4^{q'}) = v_{10}^G$, and $h(v_5^{q'}) = v_6^G$ at $\Delta_t = 5$ (around the \heartsuit marker).

A summary of the notations used throughout this paper can be found in Table 1.

3 RELATED WORK

Various applications of temporal graphs and sources of temporal graph data can be found in surveys on temporals graphs [4, 10]. Existing research on temporal graphs primarily focuses to temporal paths and their applications [1, 4, 12, 13, 15, 19, 22, 23]. None of these works study the search of a given query pattern. Most related is existing work on temporal community detection over temporal networks [9, 14, 25, 26] and multi-layer networks [3]. These work first identify communities in a static network, then identify the evolution of the communities from the changes of the network. Specifically, the problem of finding dense patterns in temporal graphs has been studied in [25, 26]. This work allows to find diversified dense regions, thus minimizing the temporal and social redundancy of the returned patterns. Such diversification may also be applied to the arbitrary patterns mined in this work, but is not in this paper’s scope. A recent approach [16] considers relations between edges, namely the time-respecting subgraph isomorphism problem, where edges are put into temporal sequence of each other. This is useful to model propagation in a network, but cannot handle more complex temporal constraints in the query. The authors propose a time-first, topology-first and a hybrid solution to approach their problem. A consideration of the fact that edges can appear and disappear over time is made in [18], which focuses on finding structural subgraph patterns in the graph that persist over the longest period of time. This is supported by three index structures that store label information, neighbourhood constellations and path maps.

In summary, these works can be used to find dense regions such as cliques and quasi-cliques in a temporal network, but do not allow to find patterns arbitrarily shaped over time. Thus, to

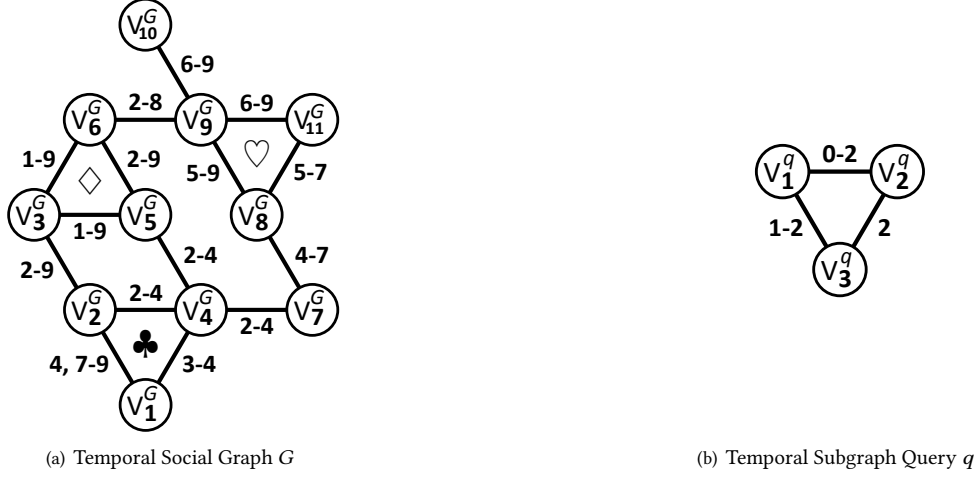


Figure 2: Example TSG and TSQ. Time points where the time dependent function of an edge returns a non-zero value are noted next to the edge: A dash (‘-’) is used to denote intervals and commas (‘,’) indicate enumeration of timepoints or intervals. The suit markers (♥, ♣, ♦) give visual guidance for the text description.

$t \in \mathcal{T} = \{0, 1, \dots, m\}$	time-domain (discrete)
$G = (V^G, E^G, F^G)$	temporal social graph G vertices/nodes V , edges/links E , time-dependant function $F^G = \{f_{v_i^G, v_k^G}^G(t) \in \{0, 1\} (v_i^G, v_k^G) \in E^G\}$ ex. Figure 2(a)
$q = (V^q, E^q, F^q)$	temporal social query graph q ex. Figure 2(b) ex. Figure 6
$S \in \mathcal{S}$ $S = (V^S, E^S, F^S)$	subgraphs of G matching q
$G^\perp = (V^G, E^G)$	non-temporal projection of G
$h : V^q \rightarrow V^S$	bijection mapping
M^S	assignment map of an isomorph S to q within G size of $M^S: V^q \times V^G $
M^0	aggregation of all M^S ex. Figure 7
$SSG = (V^{SSG}, E^{SSG})$	simple subgraph structure ex. Figure 3 ex. Figure 9
♣, ♥, ♣, ♦	visual markers

Table 1: Notations used throughout this paper.

the best of our knowledge, our work is the first one for finding patterns in temporal graphs such that the query pattern exhibits temporal constraints.

A further line of related work is pattern search on static (non-temporal) graphs. The solution to this problem was proposed by Ullmann [21] and serves as our baseline. This problem, for which a survey of solutions is found in [5], still attracts vivid research attention (e.g., [17, 20]). This pattern search has further been extended to labelled vertex-graphs, as surveyed in [8]. While this research field has received extensive attention, these solutions are not applicable to our problem setting since they do not consider any time-dependent network structures, which increases the complexity of the problem.

Furthermore, solutions have been presented for the problem of finding subgraphs in a large collection of small graphs [24]. This approach first mines frequent structures and stores for each frequent structure the IDs of the graphs that contain it (similar to an inverted file). A query can then be answered by identifying the frequent structures contained in it and intersecting the corresponding lists of IDs. Although, this problem setting is also fundamentally different, it nonetheless inspired us for the index structure proposed in this work.

4 BASIC TEMPORAL SUBGRAPH MATCHING

The problem of temporal subgraph matching is related to the classic subgraph isomorphism counting problem, which is to find the set \mathcal{S} of *all* subgraphs of a non-temporal graph G that are isomorphic to a given non-temporal query graph q . This problem is a generalization of the NP-complete subgraph isomorphism problem [6, 11], where the challenge is to decide whether *any* such subgraph $S \in \mathcal{S}$ exists in G . Consequently, the subgraph isomorphism counting problem is NP-hard, since its result can be used in to decide the subgraph isomorphism problem in $O(1)$, by testing the number of subgraphs for positivity. The current best-known algorithm for obtaining the exact count of an arbitrary query graph q is in $O(n^{\frac{\omega k}{3}})$, where k is the size of query graph q and ω is the exponent of fast matrix multiplication [7]. Our problem is at least as hard as subgraph counting, as we want to enumerate all instances of q in G , while also considering temporal constraints on edges.

The problem of finding all subgraph isomorphisms on static (non-temporal) graphs can be extended to temporal subgraphs as follows: Given a temporal query subgraph $q = (V^q, E^q, F^q)$, initialize the non-temporal graph $q^\perp = (V^q, E^q)$, where $F^{q^\perp} \neq 0$. In informal words, two nodes in q^\perp are connected iff their corresponding vertices in q are connected at least at one point of time. This can be seen as a projection of the temporal query q to a single point of time. Now this projection is applied on the temporal graph $G = (V^G, E^G, F^G)$ as well, yielding the non-temporal graph $G^\perp = (V^G, E^G)$. Now solve the subgraph isomorphism problem on the “flattened” by finding the set of all subgraphs \mathcal{S}^\perp of G^\perp that are isomorphic to q^\perp . This yields, for each any edge e in any resulting graph $S^\perp \in \mathcal{S}^\perp$ a mapping $h(e)$ to an edge in G as described in Definition 2.2.

Since the temporal subgraph query is more selective than the non-temporal query by considering temporal constraints, an additional refinement step is necessary. For any $S^\perp \in \mathcal{S}^\perp$ to be verified as a result of the overall temporal subgraph query, there must exist a time offset Δ_t such that the time-dependent function F^q matches the time dependent function of S . More formally, $S^\perp = (V^S, E^S)$ satisfies the temporal subgraph query of q iff $\exists \Delta_t \in \{0, \dots, m - q^t\} : \forall e \in S^\perp : \forall t \in [0, t^q] : f_e^q(t) = f_{h(e)}^G(t + \Delta_t)$.

4.1 Subgraph Isomorphism in Non-Temporal Graphs

Ullmann [21] introduces a viable method for solving the (non-temporal) subgraph isomorphism problem, which we will extend and briefly describe in this chapter: Let G be a non-temporal graph, which is the special case of a temporal graph having a singular time domain $\mathcal{T} = \{0\}$ and having all edges in E^G exist at time 0. For every subgraph $S = (V^S, E^S) \in G$ that is isomorph to a query graph q , we can define a $|V^q| \times |V^G|$ matrix M^S , such that $M_{i,j}^S = 1$ iff $h(v_i^q) = v_j^G$ and 0 otherwise. M^S can therefore be interpreted as an assignment map that locates the vertices of the subgraph in the larger graph. Note that in every row of M^S , there is exactly one cell with the value of 1, while in every column there is at most one cell to contain a 1.

Let furthermore M^0 be a matrix having the same dimensions as M^S and $M_{i,j}^0 = 1 - \prod_{S \in \mathcal{G}} (1 - M_{i,j}^S)$, so that M^0 gives information

about whether there exists any unspecified subgraph S so that $h(v_i^S) = v_j^G$. It is now possible to retrieve the individual subgraph matrices M^S from M^0 , along with possible other matrices, which do not belong to a valid subgraph query solution: Alter the cells of M^0 by setting different cells from 1 to 0, until the constraints of a subgraph representation are fulfilled (exactly one 1 per row, max. one 1 per column).

The main idea is to mine candidate subgraphs S from M^0 , which are matching in the “flattened” graph G^\perp that is oblivious to the time constraints. Therefore, a cell $M_{i,j}^0 = 0$ implies that there exists no subgraph S where $h(v_j^S)$ would map vertex v_j^S to vertex v_i^G in the social network. Analogously, a value of 1 implies that such a graph may possibly exist. A trivial case is to set every $M_{i,j}^0$ to 1, which means that every vertex in G can be part in any subgraph that fulfills q . But in order to improve the runtime of the algorithm, it is desirable to reduce the number of subgraph combination. This can be achieved by setting as many cells in M^0 to 0 as possible. There are different methods to prune candidates with varying complexity and efficiency:

- **Pruning based on a node’s degree.** If $\deg(v_i^S) > \deg(v_j^G)$, then $M_{i,j}^0$ can be set to 0. Note however, that in social networks the degree of the vertices is usually much higher than the degree of vertices in the query pattern q , which is why this approach yields limited pruning power.
- **Pruning based on invalid neighbour mappings.** Vertices can be pruned if there is no valid assignment for its neighbours, although the node itself can be mapped between S and q . More formally, a cell $M_{i,j}^0 = 1$ can be set to 0 (and thus be pruned), if there is a neighbour vertex v_u^S of v_i^S (i.e. $(v_i^S, v_u^S) \in E^S$) and no neighbour v_w^G of v_j^G (i.e. $(v_j^G, v_w^G) \in E^G$) such that $M_{u,w}^0 = 1$:

$$M_{ij}^0 \leftarrow M_{ij}^0 \cdot \left(\prod_{(v_i^S, v_u^S) \in E^S} \left(1 - \prod_{(v_j^G, v_w^G) \in E^G} (1 - M_{uw}^0) \right) \right)$$

Pruning a cell may allow for further pruning of other cells, so a new pruning iteration should be invoked after a successfully setting a cell to 0. This method can be stacked with other methods to further remove further candidates after another method was successful in removing candidates.

4.2 Additional Pruning Filters for Temporal Graphs

Besides the generic filter steps enlisted in Section 4.1, we furthermore introduce two more viable filter steps that can be applied in the context of *temporal* subgraph isomorphisms:

- **Pruning based on time offset.** As described before, every derived subgraph S that is a candidate to be isomorph to the query (represented through M^S) needs to be refined in the sense that there needs to exist a Δ_t so that the time-dependant functions match. When testing various Δ_t , it is feasible to create a copy $M_{\Delta_t}^S$ of M^S . Since only the time frame from Δ_t until $\Delta_t + t^q$ is relevant for the matching of the functions, the graph G can be projected onto a more sparse graph N_{Δ_t} than N by only inserting an edge into

$E^{\Delta t}$, if the corresponding edge in G exists in this more narrow time frame.

- **Pruning based on network distance.** When iterating through M^0 , after setting a candidate value for the first processed row of M^0 , we can try to cut down the number of columns that can contain 1s at all. Let i be the index of a row where exactly one column j is set to one. Then we can compute the maximum hops from v_i^S to any other node in S . Then we determine all nodes in the graph N , whose hops distance to v_j^N is larger than that distance. Those columns can then be set to zero. This is a viable approach if $|V^N| \gg |V^S|$ and N is sparse. For efficiency reasons, it is recommended to pick the first row i in a way so that v_i^S lies in a *central* position in S , e.g. minimizing the maximum hops distance. As temporal aspects are not a pruning criterion for this filter, it can generally be applied to non-temporal subgraph isomorphism queries as well.

In our experiments, we will take a deeper look at the effectiveness and performance of the basic and our extended filters. We will also evaluate the processing order, in which these filters are applied.

5 AN INDEX STRUCTURE FOR TEMPORAL GRAPHS

In this section we will give an in depth description of how to build an index for temporal social graphs and how to perform pattern queries on this index.

The construction of an index structure that supports subgraph pattern search on temporal graphs can be summarized in four steps: (1) Select one or more simple subgraph structures (SSGs) and do the following steps for each of them. (2) Find each occurrence of the SSG in the graph G without consideration of the temporal aspect. (3) Transform each occurrence into a string reflecting its unique behaviour over time considering the functions f of the edges. (4) Index the obtained strings using an index structure for substring search. In the following we will consider each of these steps in detail.

Simple Subgraph Structure Selection: The selection of suitable SSGs ($SSG = (V^{SSG}, E^{SSG})$) is crucial for the performance of the index, since the index can later on only answer queries that contain at least one of the selected SSG. A good set of SSGs should thus contain even the simplest possible query structures. Let us note that a temporal pattern query on a TSG must involve at least a relationship (edge) of two entities (nodes). The most simple SSGs involving 2 and 3 nodes, illustrated in Figure 3, should thus always be indexed in order to allow index support for all possible queries. When challenging the trade off between simplicity and ubiquitousness of SSGs, multiple different SSGs may be indexed in parallel to suit a wider array of queries. In the following we will showcase the construction of the index based on the triangle structure.

Finding SSG Occurrences: To find all occurrences of the SSG in the graph, the temporal aspect of G will be neglected thus using “flattened” version of G^\perp of G as used in Section 4. Within G^\perp , we search for all occurrences of the SSG using a traditional subgraph isomorphism algorithm such as [21]. An SSG occurs at a set of nodes $V^O \subset V^G$ iff $(v_i^O, v_j^O) \in E^N \Leftrightarrow (v_i^{SSG}, v_j^{SSG}) \in E^{SSG}$. Please note that due to possible symmetries, several occurrence

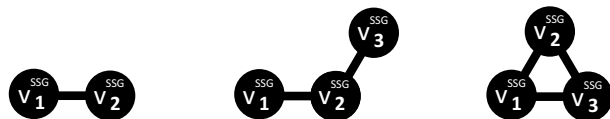


Figure 3: Simple Subgraph Structures (SSGs) with 2 and 3 nodes

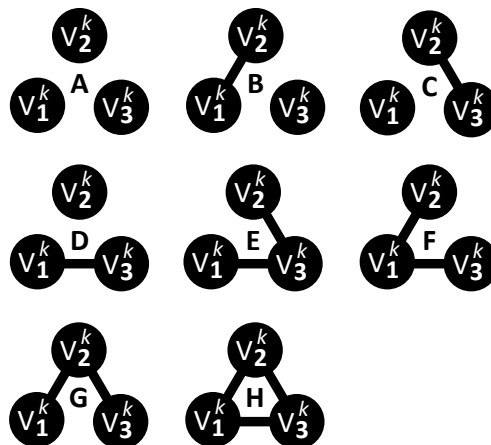


Figure 4: Encoding the edges of a graph. Each possible graph topology is encoded by an uppercase Latin character A-H.

can happen for the same set of V^O , depending on how V^O is mapped to V^{SSG} .

Example 5.1. The triangle SSG occurs within the graph of Figure 2(a) at the positions marked with \heartsuit , \clubsuit , and \diamond . If there would be an edge (v_6^G, v_{10}^G) at time 1, the set of nodes v_6^G, v_9^G, v_{10}^G would also form an occurrence, even though at no point of time an actual triangle is formed between the nodes.

The identification of those SSG occurrences does not come free of cost: In particular, finding all subgraphs of G that are an occurrence of the SSG has a runtime complexity of $O(n^{\frac{\omega k}{3}})$, where k is the size of query graph q and ω is the exponent of fast matrix multiplication [7]. However, this task can be performed offline and will not affect the query performance. Furthermore, the actual runtime is in general low enough due to two reasons: First, the SSGs are usually very small (in our example not more than 3 nodes) and second, the graph G is not fully connected in a real-world setting.

String Transformation: At every discrete point of time, a set of nodes in the graph that belong to an SSG can form a certain constellation via their edges. Figure 4 shows all possible combinations for a triangle SSG consisting of three, distinguishable nodes (v_1^{SSG}, v_2^{SSG} , and v_3^{SSG}) and assigns each constellation a unique symbol; here we are using uppercase Latin characters. To encode an SSG’s temporal behaviour over time, at each time frame the currently present edges have to be figured out and be mapped via a pre-defined assignment table (such like Figure 4) to a unique symbol or character. In general, the alphabet to encode all constellations of an SSG having k edges consists of 2^k

characters. Concatenating these characters along the chronological time-series will yield in a string representing the temporal behaviour.

Example 5.2. The graph q denoted in Figure 2(b) shall be represented by a string using the triangle SSG. When each node v_i^q is mapped to v_i^{SSG} , the symbol representing the graph constellation at each time frame t_j ($0 \leq j \leq t^q$) can be looked up in Figure 4 ($t_0: B, t_1: F, t_2: H$), thus yielding the string BFH . For other possible mappings of v^q to v^{SSG} the string representations $BGH, CEH, CGH, DEH,$ and DFH are valid as well.

This schema can be applied to all substructure occurrences found in the graph, so that each occurrence's temporal behaviour can be described through a string. It is then feasible to index those strings in a way to efficiently support substring search. We propose to employ a suffix-tree to index these substrings concisely.

Example 5.3. Consider the triangle SSG. It occurs three times in our example graph G (Figure 2(a)), namely at $\heartsuit, \clubsuit,$ and \diamond . For every occurrence, there exist six possible permutations of how the substructure can occur at this position, due to the ways v_i^G may be mapped to v_i^{SSG} . We depict all of these occurrences and permutations in Figure 5.

6 QUERY PROCESSING

Next, we describe how our string-index can find all occurrences of a given temporal query pattern q . As described in Section 5, in the following, we assume that an index has been build for a specific simple subgraph structures SSG.

- (1) Identify occurrences of the SSG in the "flattened" temporal graph query q^\perp .
- (2) For each such occurrence, perform the same string transformation than performed for the index (i.e., use the same character map).
- (3) Index-supported search for the transformed string to find candidates for verification.
- (4) Refine the candidates through verifying that the part of q which is not contained in the SSG is isomorph to the surrounding of a candidate.

In more detail, to answer queries according to Definition 2.2 using the index support of the suffix-tree, we first have to isolate those SSG occurrences in the graph topology of the query graph of the SSG that was used for the string transformation process before. An SSG may occur not at all, once, or multiple times in the graph. If no SSG occurrence can be found, the index is of no help and the search has to default to a full scan, which is why there is a motivation to keep SSGs small and simple. In case of one or several occurrences of the SSG in q , we isolate the temporal behaviour of that part of the query graph and transform it using the same string encoding method used for the index construction.

Since the queried time frame is usually smaller than the indexed time span, the length of the string derived from the SSG occurrence in q is shorter than the length of string belonging to the occurrence in the graph which is stored in the index. To answer the query, we now must find all those strings in the index that contain the substring belonging to the query.

Example 6.1. Identification and String Transformation of SSG in query q . In our example query q' (Figure 6), the triangle SSG occurs at the \clubsuit marker. Since there are six possible permutations of the occurrence, valid string transformations are $EHH, EHH, FHH, FHH, GHH,$ and GHH . With each of these unique query strings, we can search our encoding index (suffix tree) for entries that contain the query string. The substring positions are indicated through bold and underlined text in Figure 5. Entries which do not contain one of the substrings can be pruned.

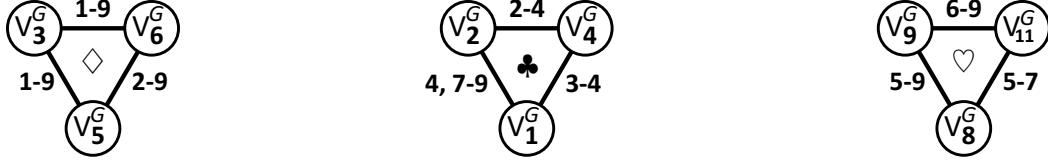
Since practical SSGs consist of more than one node, there is usually always more than one way it can be mapped to either the nodes of the occurrence in the query graph or the occurrence in the main. I.e., there are also several string transformations of the occurrence. There are in general several ways to approach these permutations:

- Account for permutations at index creation and query processing. This means that every permutation is indexed, thus resulting in a larger index, and that index is queried multiple times (once for each permutation of the query)
- Consider the permutations in the index, but not for the query
- Consider the permutations only within the query
- Neglect the permutations in the query graph and at index creation.

Neglecting all permutations may result in correct results not being found, as there is no guarantee that an 'identical' permutation has been used for the index and the query. On the other hand, if permutations are considered both times, the result will also show how the query 'fits' into the graph, i.e. the direct mapping from the nodes can be deducted. However, multiple queries have to be performed on an increased index, thus increasing query cost. As a trade-off, it is possible to only consider all permutations on one side (either the index *or* the query), and then find out the mapping in a refinement step. We recommend to consider all permutations within the index and not for the query, as submitting multiple queries increases the overall query cost linearly, while linearly enlarging the index results in a sublinear increase in query performance. Compared to the approach where permutations are considered on both sides, a refinement step is now necessary to deduct the exact mapping of the query to the substructure (one mapping per possible permutation). This is likely to be done faster after the query than it is to do multiple queries (one for each permutation).

Example 6.2. Index-supported Search for Transformed String Representation of the SSG. If following our advice to only consider permutations within the index, querying our example query q' (Figure 6) with a triangle SSG, the search string will either be EHH **or** FHH **or** GHH . The exemplary index in Figure 5 will then yield the set of nodes $\{v_3^G, v_5^G, v_6^G\}$ for the \diamond , and the set $\{v_8^G, v_9^G, v_{11}^G\}$ for the \heartsuit occurrence. However, the mapping of $v_i^{q'}$ to v_j^G **cannot** be deducted from the index and has to be refined computationally.

Searching for a substring in the index then retrieves two important facts for every match: (1) a subset of V^G that corresponds to an SSG occurrence, and (2) the temporal offset Δ_t at which it occurred (calculated by the offset position of the substring from the beginning of the indexed string). Both are crucial for



Mapping			String	Mapping			String	Mapping			String
v_1^{SGG}	v_2^{SGG}	v_3^{SGG}	Transformation	v_1^{SGG}	v_2^{SGG}	v_3^{SGG}	Transformation	v_1^{SGG}	v_2^{SGG}	v_3^{SGG}	Transformation
v_3^G	v_5^G	v_6^G	AF HHHHHHHH	v_1^G	v_2^G	v_4^G	AACEHAABBB	v_8^G	v_9^G	v_{11}^G	AAAA AF HGG
v_3^G	v_6^G	v_5^G	AF HHHHHHHH	v_1^G	v_4^G	v_2^G	AACGHAADDD	v_8^G	v_{11}^G	v_9^G	AAAA AF HGG
v_5^G	v_3^G	v_6^G	AG HHHHHHHH	v_2^G	v_1^G	v_4^G	AADEHAABBB	v_9^G	v_8^G	v_{11}^G	AAAA AG HGG
v_5^G	v_6^G	v_3^G	AE HHHHHHHH	v_2^G	v_4^G	v_1^G	AABGHAFFFF	v_9^G	v_{11}^G	v_8^G	AAAA AE HGG
v_6^G	v_3^G	v_5^G	AG HHHHHHHH	v_4^G	v_1^G	v_2^G	AADFHAACCC	v_{11}^G	v_8^G	v_9^G	AAAA AG HGG
v_6^G	v_5^G	v_3^G	AE HHHHHHHH	v_4^G	v_2^G	v_1^G	AABFHAACCC	v_{11}^G	v_9^G	v_8^G	AAAA AE HGG

Figure 5: Transforming the temporal behaviour of the triangular SSG occurrences in G to strings.

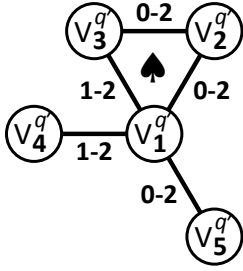


Figure 6: A more complex TSQ q' .

efficiently refining the candidates. A refinement is necessary, as the query may be more restrictive than the SSG itself, i.e. through additional edges attached to the SSG, the retrieved candidates are a superset of the results. Therefore, this set has to be refined, i.e. it has to be checked whether the found SSG is part of a larger subgraph structure that can fulfill the query constraints with regards to graph structure as well as temporal behaviour. To refine the candidates, we return to the concept of Section 4.1. Before evaluating the substring candidates, we can still apply the degree filter for q' and G to M^0 , as it is a quick way to eliminate some impossible assignments.

Example 6.3. Figure 7 shows M^0 for our sample query q' after applying the degree and neighbour filters. It shows that for example $v_1^{q'}$ can only be mapped to nodes with a degree of at least 4, thus leaving only cells $M_{1,4}^0$ and $M_{1,9}^0$ in the first row with a 1. On the other hand, $M_{2,3}^0$ is set to 0, as $v_2^{q'}$ cannot be mapped to v_4^G , as v_4^G (unlike $v_2^{q'}$) does not have a neighbour of degree 4.

After optimizing M^0 , for each found SSG x , we initialize a copy of M^0 denoted as M^x and thereby set $M_{i,j}^x := M_{i,j}^0$. Each found SSG instance x gives us a 'hint', where an occurrence of q' in G may occur as well as the temporal offset Δ_t at which the temporal pattern of the subgraph structure matched. This hint will either lead to a correct result or may be false – but

M^0	v_1^G	v_2^G	v_3^G	v_4^G	v_5^G	v_6^G	v_7^G	v_8^G	v_9^G	v_{10}^G	v_{11}^G
$v_1^{q'}$	0	0	0	1	0	0	0	0	1	0	0
$v_2^{q'}$	1	1	0	0	1	1	1	1	0	0	1
$v_3^{q'}$	1	1	0	0	1	1	1	1	0	0	1
$v_4^{q'}$	1	1	0	0	1	1	1	1	0	1	1
$v_5^{q'}$	1	1	0	0	1	1	1	1	0	1	1

Figure 7: Assignment matrix M^0 for q' and Q after applying the degree and neighbour filters.

M^{q_4}	v_1^G	v_2^G	v_3^G	v_4^G	v_5^G	v_6^G	v_7^G	v_8^G	v_9^G	v_{10}^G	v_{11}^G
$v_1^{q'}$	0	0	0	1	?!	0	0	0	0	0	0
$v_2^{q'}$	0	0	?!	0	0	0	0	0	0	0	0
$v_3^{q'}$	0	0	0	0	0	1	0	0	0	0	0
$v_4^{q'}$	1	1	0	0	0	0	1	1	0	1	1
$v_5^{q'}$	1	1	0	0	0	0	1	1	0	1	1

Figure 8: Assignment matrix M^{q_4} after applying the assignment from the index candidate. This means that the bold cells should be set to 1. However, cells depicted with '?!' already contain a 0 (ref. Figure 7) and cannot be set to 1 in a valid way.

its correctness will not have any effect on other SSG instances, which is why we can process them individually and in parallel. Since the index found a matching SSG which implicitly matches

the vertices of the structure to vertices of the graph (after considering the permutations of the mapping), we can assign $|V^{SSG}|$ fix assignments, thus nullifying all other cells in those rows (since M^X is usually much wider than tall, this drastically increases the number of cells containing a 0). Since Δ_t is known at this point of time, we can now apply the more sophisticated time offset and network distance filters (Section 4.2) to M^X .

Example 6.4. Let us return to our running example, where we search for occurrences of the more complex query q' (Figure 6) in the running social network G (Figure 2(a)). Let's consider the fourth permutation of the \diamond substructure, which is returned as a candidate through the query permutation EHH ($v_1^{q'} \rightarrow v_1^{SSG}$, $v_2^{q'} \rightarrow v_3^{SSG}$, $v_3^{q'} \rightarrow v_2^{SSG}$) (see Figure 5). Mapping this SSG to G yields the mappings $v_1^{q'} \rightarrow v_1^{SSG} \rightarrow v_5^G$; $v_2^{q'} \rightarrow v_3^{SSG} \rightarrow v_3^G$; and $v_3^{q'} \rightarrow v_2^{SSG} \rightarrow v_6^G$ also shown in Figure 5. We attempt to make this assignment in the corresponding matrix $M^{\diamond 4}$ depicted in Figure 8. Therefore, we have to set the values in $M_{1,5}^{\diamond 4}$, $M_{2,3}^{\diamond 4}$ and $M_{3,6}^{\diamond 4}$ to 1, but we see that the first two entries already contain a zero in the global assignment matrix M^0 , such that this assignment does not yield a valid matching. Therefore we can stop here and prune this candidate. In fact, in our example we can prune all permutations of \diamond in the same way, as well as permutations 1, 3, 5, and 6 of \heartsuit ; with \clubsuit not even providing candidates. In summary, this just leaves $M^{\heartsuit 2}$ and $M^{\heartsuit 4}$ for further refinement.

We are now left with a set of matrices that we need to derive final assignment candidates from. A naive way would be to iterate over all possible assignments and verify them; one would do that by choosing a single $M_{i,j}^X = 1$ and using it as an assignment, thus nullifying other values row i and column j , and then proceeding to row $i + 1$, re-applying the same concept. This needs to be done iteratively for all $M_{i,j}^X = 1$ of a row. We aim to improve this exponentially expensive approach by using heuristics. Therefore, we first take effort in finding a clever order of which we process the rows. We process rows in breadth-first-order starting at the corresponding SSG occurrence, skipping lines having a "1" assigned by the SSG occurrence. For any other line b , we look at any previous row a such that $(v_a^{q'}, v_b^{q'}) \in E^{q'}$, i.e., a neighbour of $v_b^{q'}$ that we have already assigned. Since we always start with a pre-assigned row and proceed in a breadth-first-manner, such a row must exist. That row a contains a single assignment $h(v_a^{q'}) = v_c^G$. Assuming this assignment is correct, we only need to look at columns d where v_d^G is a neighbour of v_c^G , thus having $(v_c^G, v_d^G) \in E^G$. This can be deduced from the following:

$$\begin{aligned} h(v_a^{q'}) = v_c^G \wedge h(v_b^{q'}) = v_d^G &= v_c^G \wedge (v_a^q, v_b^q) \in E^q \\ &\Rightarrow (v_c^G, v_d^G) \in E^G \end{aligned}$$

because $E^q \subseteq E^G$.

For every neighbour v_d^G where $M_{b,d}^X = 1$ and the temporal patterns match, we create a copy of the current M^X , nullify all other cells in row b and proceed to the next row in M^X . When the last row is reached, every vertex in v^q has been assigned exactly one partner in v^G , thus being a result.

Example 6.5. For our running example, we retrieve the following assignments from $M^{\heartsuit 2}$ and $M^{\heartsuit 4}$:

$$M^{\heartsuit 2} : h(v_1^{q'}, v_2^{q'}, v_3^{q'}) = (v_9^G, v_8^G, v_6^G)$$

$$M^{\heartsuit 4} : h(v_1^{q'}, v_2^{q'}, v_3^{q'}) = (v_{11}^G, v_8^G, v_9^G)$$

The later one is invalid, as $M_{1,11}^0$ is already 0 and $v_1^{q'}$ can therefore not be mapped to v_{11}^G . Following the first assignment, we now retrieve two final candidates for the complete occurrence of q' :

$$C_1 : h(v_1^{q'}, v_2^{q'}, v_3^{q'}, v_4^{q'}, v_5^{q'}) = (v_9^G, v_8^G, v_{11}^G, v_6^G, v_{10}^G)$$

$$C_2 : h(v_1^{q'}, v_2^{q'}, v_3^{q'}, v_4^{q'}, v_5^{q'}) = (v_{11}^G, v_8^G, v_9^G, v_{10}^G, v_6^G)$$

of which the first one C_1 is invalid, as the time-dependant-function on edges $(v_1^{q'}, v_4^{q'})$ and (v_9^G, v_6^G) do not match. C_2 is a valid result to the query.

7 EXPERIMENTS

In this section we show experimental results of our proposed methods. As a baseline approach, we resort to Ullmann's algorithm as described in Section 4.1. This represents the expansion of traditional solutions to the temporal domain. We further evaluate the included filters as well as our proposed additional filters individually and in combination to distinguish the naive baseline approach from a more advanced setup. We then compare this baseline approach, which has been extended to temporal graphs, to our advanced query processing approach proposed in Section 6 supported by the index structured introduced in Section 5. Additionally, we introduce the evaluated queries and the employed datasets. All experiments were performed on a 3Ghz workstation having 32 GB of RAM. The experiments were run on a single core.

7.1 Datasets

As a datasource for our real data evaluation we use a snapshot of the ACM Digital Library² taken on Dec 15, 2014 consisting of 582,150 publications with author information. Using only the co-author relationship for each calendar year, we build a temporal social graph $G = \{V^G, E^G, F^G\}$, reflecting the collaboration network over time, in the following way:

- Each researcher present in the dataset is represented by a node v^G .
- Two researcher nodes v_i^G and v_j^G are connected by an edge (v_i^G, v_j^G) if they have at least co-authored one publication at any time $t \in \mathcal{T}$.
- The time dependent function for an edge (v_i^G, v_j^G) indicates collaboration over time and is set the following way:

$$f_{v_i^G, v_j^G}^G(t) = \begin{cases} 1 & \text{if } v_i^G \text{ and } v_j^G \text{ co-authored a publication in year } t \\ 0 & \text{if } v_i^G \text{ and } v_j^G \text{ did not co-author a publication in year } t \end{cases}$$

The resulting temporal social graph is called *PUBS* in the remainder of this section. In order to evaluate naive approaches of the proposed algorithms we also use small subgraphs of *PUBS*

²<http://dl.acm.org/>

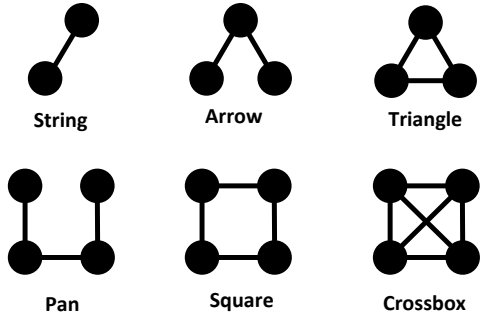


Figure 9: Evaluated subgraph structures in our experiments along with assigned names.

	PUBS	MiniPUBS	MicroPUBS
Nodes	379,188	10,000	3,792
Edges	2,114,720	77,568	36,548
Timepoints	69	46	43
# Strings	2,114,720	77,568	36,548
# Arrows	44,379,646	1,541,152	917,810
# Triangles	13,191,264	422,736	176,286
# Squares	449,684,160	5,179,608	2,259,456
# Pans	1,438,921,874	25,601,204	21,972,438
# Crossboxes	411,978,792	3,967,824	1,196,184

Table 2: Characteristics of the researcher collaboration datasets

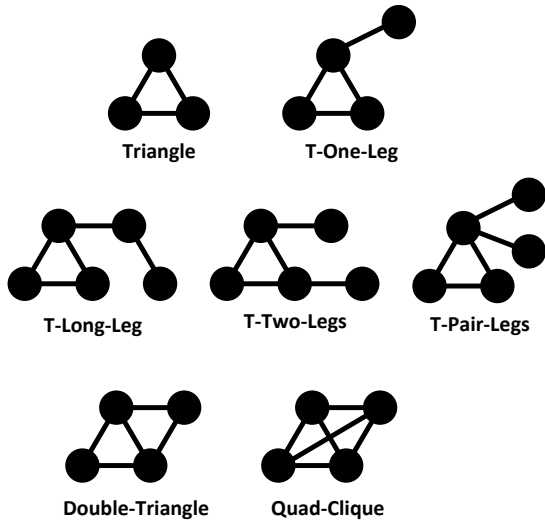


Figure 10: Small queries used in our experiments.

called *MiniPUBS* and *MicroPUBS*, with 10000 nodes and 3792 nodes, respectively. These subgraphs were generated from PUBS by performing a breadth-first search rooted at the first two (anonymous) authors of this work.

Table 2 summarizes the characteristics of the three datasets. In addition to the number of nodes, the number of edges and the duration of the network in years Table 2 contains the number of the subgraph structures illustrated in Figure 9.

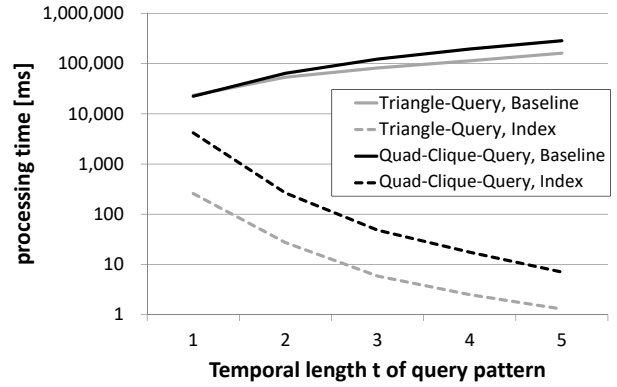


Figure 11: Querying with a triangle and a crossbox structure of constant edges and variable length t^q .

7.2 Queries

Figure 10 shows a set of standard query subgraphs that we used in our experimental evaluation. In the first set of experiments, the query time domain t^q is set to relatively small values of $3 \leq t^q \leq 5$, to all the baseline approaches to terminate in reasonable time. The temporal pattern on these subgraphs is chosen uniformly random, such that at each point of time $t \in \mathcal{T}^q$ of the query graph, any edge has a chance of 50% to be part of the query pattern. To avoid degenerated cases, we ensure that each edge of a standard query is required to exist at at least one time t .

7.3 Baseline vs. Index

In a first experiment we compare the performance of our proposed index structure (cf Section 5) with the baseline approach as discussed in Section 4. Since the baseline approach is very time consuming we did this experiment on the *MicroPUBS* dataset. As a basic subgraph structure for our index we used the triangle and as queries we evaluated the triangle-query and the quad-clique-query with increasing temporal query length t^q . The results are shown in Figure 11. The baseline approach has to build a projection of the original TSG for each possible start time t over the duration of the query. With increasing duration of the query, this projection becomes more and more dense, which results in increasing runtime. The index based query processing on the other hand performs much faster in this setting. Note that, although the triangle query is beneficial to the index (since the index is built on the triangle structure), the quad-clique query can also be answered efficiently. With increasing query duration, the results quickly decrease, yielding a lower number of candidates, which leads to even lower runtime.

Figure 12 shows the query time for various query patterns. As most of the 4-year-long queries have a highly specific temporal pattern, the index-based approach profits from early pruning of large parts of the data, while the baseline approach is first looking for the general graph structure and can only prune at a second step where the temporal behaviour is considered. We see that the *Triangle* query for $t^q = 3$ has the highest run-time using our index, while having the lowest run-time of the baseline. The reason is that this query yields the largest number of (verified) results which, trivially, cannot be pruned. When changing the query time t^q for the more complex *T-Two-Legs* query, we can

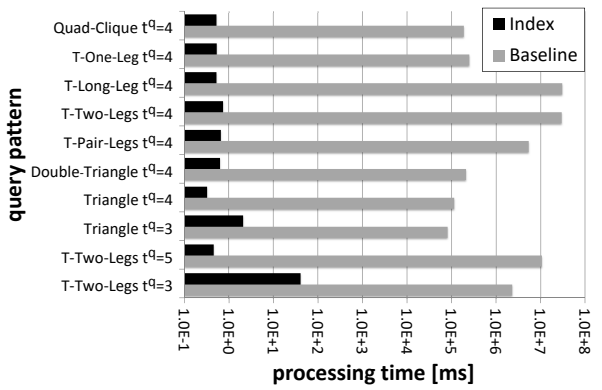


Figure 12: Querying with a distinct queries.

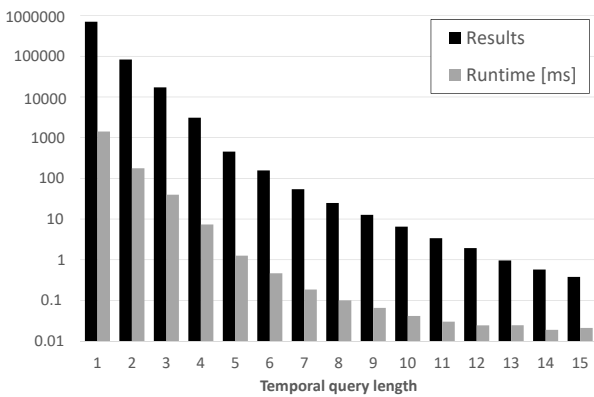


Figure 13: no. of results and runtime with increasing query length

again observe that our index supported approach can benefit from early pruning. Note that the time needed to build the index for this microPUBS dataset was less than a second.

7.4 Evaluating query parameters

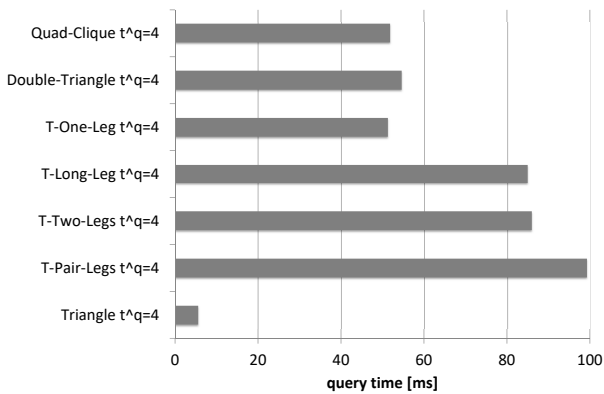


Figure 14: query time for harder queries

In the next set of experiments we demonstrate the effect of the query duration t^q on the large PUBS dataset. The *Baseline* approach was not evaluated on this dataset due to its excessive run-time. In the first experiment, shown in Figure 13, we use the

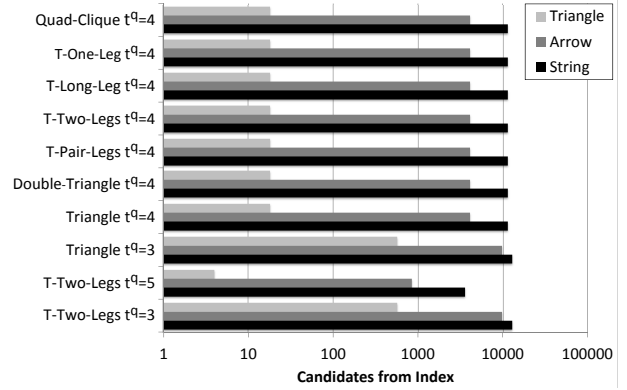


Figure 15: Testing different indexed subgraph structures: number of candidates

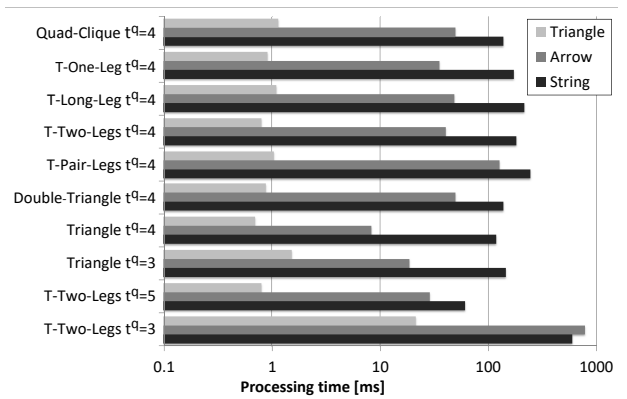


Figure 16: Testing different indexed subgraph structures: processing time

Triangle structure for query and indexing. Index construction on this large dataset took less than 30 minutes. Results using this index are averaged over 1K runs for random temporal patterns of the PUBS dataset. We observe that the run-time is directly proportional to the number of results. This behaviour is expected, since less refinement is needed.

The previous experiment show-cased a best-case scenario, where the structure used for indexing is identical to the query structure. In this case, candidates returning by the index need only to be verified for temporally matching the query pattern.

To show the behaviour in more realistic scenarios, we made the topological structure of the query more complex by adding additional edges, while still using triangle SSGs for indexing. Thus, those added edges are not covered by the index and need to be accounted for in the refinement step. Figure 14 shows that adding additional edges and nodes to the query increases the processing time: Adding two edges to the triangle produces a more complex query than just adding one (than adding none), and a Quad-triangle is more specific than a Double-Triangle than a simple triangle.

Thus, in the next set of experiments shown in Figures 15 and 16, we test the efficiency of different SSGs on the MiniPUBS dataset. Here, we compare different structures used to build the index (specifically, the SSGs *Triangle*, *Arrow* and *String*), and different query structures for different time lengths t^q .

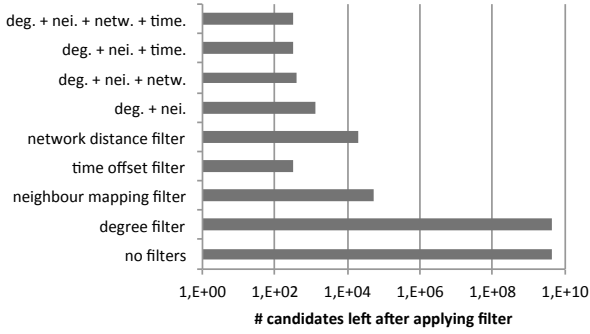


Figure 17: Comparing the number of candidates left after applying filters to M^0 .

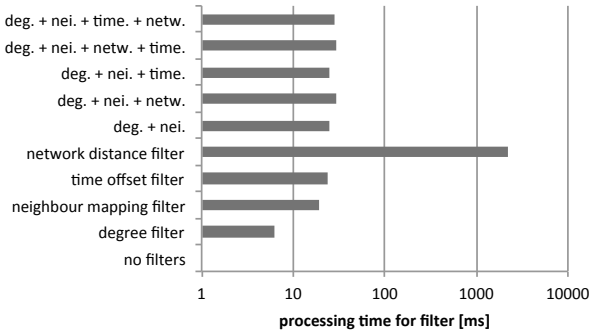


Figure 18: Measuring the computational cost of applying filters to M^0 .

Comparing these results to Table 2, we notice that simpler subgraph structures appear more often in graphs, whereas more complex structures appear less and are thus generally more selective when used as the basis for an index. This also reflects in the number of candidates produced by our index-based approach for different basic query structures. We note that in these queries, using triangle structure for indexing yields much less candidates for refinement. This is because many *Arrow* and *String* structures may not be part of a triangle, thus creating additional candidates to be pruned. However, it should be noted that the triangle index is only applicable if the query structures contains at least one triangle, which is the case for query structures featured in this experiment.

7.5 Evaluating Pruning Filters

In Section 4.2 we introduced additional pruning filters applicable to temporal graphs. We evaluated the various filters on the candidates retrieved after querying the index with a simple subgraph structure (SSG) for our running query example q' on the MiniPUBS graph. Therefore, we measure the performance of different pruning strategies (and their combinations) in their number of candidates generated from M^0 that need to be refined. Figure 17 shows that the degree filter has no effect, because the only nodes not matched in the initial assignment M^0 are the two legs not part of the SSG which just have a degree of 1. The network distance filter and the neighbour mapping filter reduce the number of candidates by a factor of about 10, 000 and 30, 000, respectively. The time offset filter has the highest pruning power, reducing the number of candidates to less than

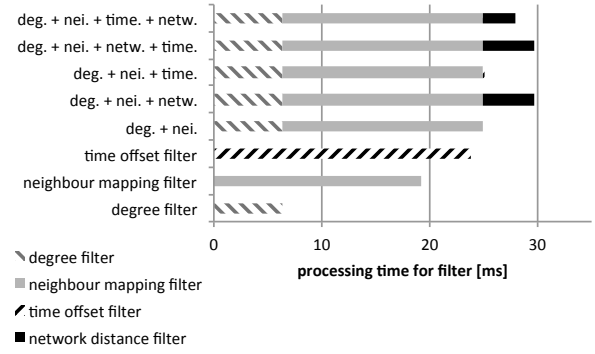


Figure 19: Measuring the computational cost of applying filters to M^0 with highlight of individual cost.

two hundred. Furthermore, variations of sequential filter combinations are depicted as well: A combination allows to narrow down the candidate size even more. Figure 18 shows the corresponding computational time of the filters: the network distance filter is by far the most expensive one, even though it cannot outperform the time filter. We contribute the bad performance of the network distance filter to the dataset, in which network distances are generally very short, as the dataset was generated by a breadth-first search of a larger network. A more detailed look into the combination of filters is shown in 19, where it becomes clear that an expensive, but selective filter like the time offset filter, becomes cheap if applied after another more generic filter and in combination gives great results and a very small candidate set. We summarize that the time-offset filter is the most powerful pruning step in our setting. This is because most candidates do not match the specific temporal patterns exhibited by the temporal query graph, such that temporal pruning is very powerful. This also shows how traditional pruning methods only, which do not consider temporal patterns, are not sufficient to efficiently find patterns on temporal graphs.

8 CONCLUSIONS

We proposed first solutions for the problem of searching patterns on temporal social networks. For this problem, existing solutions for graph isomorphism can not be applied directly, since temporal conditions need to be handled. As a baseline approach, we define a temporal pruning heuristic to augment an existing subgraph isomorphism search algorithm. Due to the high run-time of such approach on real social networks, we proposed a data structure to index all occurrences of basic graph structures, to find a candidate set of isomorphic subgraphs quickly at query time. This data structure transforms temporal graphs and temporal graph queries in strings and employs a suffix tree to organize these strings efficiently. Our experimental evaluation shows that this index structure can reduce the run-time of searching small temporal query patterns by orders of magnitude. Still challenges remain open: since the problem of isomorphic subgraph search is exponentially hard in the size of the query graph, we cannot hope to scale to large query graphs. Thus, approximate solutions are required for larger and more complex query patterns. Further, we can relax the problem to estimation of the number of isomorphic subgraphs, rather than returning all occurrences and their location in the graph. This relaxation may allow more efficient

approximations. Another important future aspect of this work is the diversification of patterns as studied in [25, 26] for dense subgraphs only. Applying such diversification to arbitrary subgraphs is a non-trivial task, as the notion of social and temporal overlap has to be redefined.

As you have seen in our running examples, temporal subgraph queries need to be defined very detailed, i.e., each configuration at every point of time needs to be stated. In cases where such level of detail is unneeded (for example when a certain link may or not exist as well at a specific point of time) our algorithm cannot specifically account for this fact. While intuitively, allowing more configurations makes queries less hard to verify, it actually increases the query complexity as the program must now consider several possible configurations instead of one. A future version could benefit in these scenarios.

REFERENCES

- [1] Miguel Araujo, Stephan Günnemann, Spiros Papadimitriou, Christos Faloutsos, Prithwish Basu, Ananthram Swami, Evangelos E. Papalexakis, and Danai Koutra. 2016. Discovery of "comet" communities in temporal and labeled graphs (Com2). *Knowl. Inf. Syst.* 46, 3 (2016), 657–677. DOI: <http://dx.doi.org/10.1007/s10115-015-0847-2>
- [2] Peter S Bearman, James Moody, and Katherine Stovel. 2004. Chains of affection: The structure of adolescent romantic and sexual networks. *American journal of sociology* 110, 1 (2004), 44–91.
- [3] Brigitte Boden, Stephan Günnemann, Holger Hoffmann, and Thomas Seidl. 2012. Mining coherent subgraphs in multi-layer graphs with edge labels. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 1258–1266.
- [4] Arnaud Casteigts, Paola Flocchini, Walter Quattrociocchi, and Nicola Santoro. 2012. Time-varying graphs and dynamic networks. *International Journal of Parallel, Emergent and Distributed Systems* 27, 5 (2012), 387–408.
- [5] Donatello Conte, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2004. Thirty years of graph matching in pattern recognition. *International journal of pattern recognition and artificial intelligence* 18, 03 (2004), 265–298.
- [6] Stephen A Cook. 1971. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*. The MIT Press, 151–158.
- [7] Friedrich Eisenbrand and Fabrizio Grandoni. 2004. On the complexity of fixed parameter clique and dominating set. *Theoretical Computer Science* 326, 1-3 (2004), 57–67.
- [8] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, and Yinghui Wu. 2011. Adding regular expressions to graph reachability and pattern queries. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*. IEEE, 39–50.
- [9] Mao-Guo Gong, Ling-Jun Zhang, Jing-Jing Ma, and Li-Cheng Jiao. 2012. Community detection in dynamic social networks based on multiobjective immune algorithm. *Journal of Computer Science and Technology* 27, 3 (2012), 455–467.
- [10] Petter Holme and Jari Saramäki. 2012. Temporal networks. *Physics reports* 519, 3 (2012), 97–125.
- [11] Richard M Karp. 1972. Reducibility among combinatorial problems. In *Complexity of computer computations*. Springer, 85–103.
- [12] Gueorgi Kossinets, Jon Kleinberg, and Duncan Watts. 2008. The structure of information pathways in a social communication network. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 435–443.
- [13] Maxim Likhachev, Dave Ferguson, Geoff Gordon, Anthony Stentz, and Sebastian Thrun. 2008. Anytime search in dynamic graphs. *Artificial Intelligence* 172, 14 (2008), 1613–1643.
- [14] Yu-Ru Lin, Yun Chi, Shenghuo Zhu, Hari Sundaram, and Belle L Tseng. 2008. Facetnet: a framework for analyzing communities and their evolutions in dynamic networks. In *Proceedings of the 17th international conference on World Wide Web*. ACM, 685–694.
- [15] Raj Kumar Pan and Jari Saramäki. 2011. Path lengths, correlations, and centrality in temporal networks. *Physical Review E* 84, 1 (2011), 016105.
- [16] Ursula Redmond and Pádraig Cunningham. 2016. Subgraph Isomorphism in Temporal Networks. *CoRR* abs/1605.02174 (2016). <http://arxiv.org/abs/1605.02174>
- [17] Carlos R Rivero and Hasan M Jamil. 2016. Efficient and scalable labeled subgraph matching using SGMATCH. *Knowledge and Information Systems* (2016), 1–27.
- [18] Konstantinos Semertzidis and Evaggelia Pitoura. 2016. Durable graph pattern queries on historical graphs. In *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*. IEEE, 541–552.
- [19] John Tang, Mirco Musolesi, Cecilia Mascolo, and Vito Latora. 2010. Characterising temporal distance and reachability in mobile and online social networks. *ACM SIGCOMM Computer Communication Review* 40, 1 (2010), 118–124.
- [20] Ha-Nguyen Tran, Jung-jae Kim, and Bingsheng He. 2015. Fast subgraph matching on large graphs using graphics processors. In *International Conference on Database Systems for Advanced Applications*. Springer, 299–315.
- [21] J. R. Ullmann. 1976. An Algorithm for Subgraph Isomorphism. *J. ACM* 23, 1 (Jan. 1976), 31–42. DOI: <http://dx.doi.org/10.1145/321921.321925>
- [22] Huanhuan Wu, James Cheng, Silu Huang, Yiping Ke, Yi Lu, and Yanyan Xu. 2014. Path problems in temporal graphs. *Proceedings of the VLDB Endowment* 7, 9 (2014), 721–732.
- [23] B Bui Xuan, Afonso Ferreira, and Aubin Jarry. 2003. Computing shortest, fastest, and foremost journeys in dynamic networks. *International Journal of Foundations of Computer Science* 14, 02 (2003), 267–285.
- [24] Xifeng Yan, Philip S. Yu, and Jiawei Han. 2004. Graph Indexing: A Frequent Structure-based Approach. In *Proc. SIGMOD*. 335–346.
- [25] Yi Yang, Da Yan, Huanhuan Wu, James Cheng, Shuigeng Zhou, and John CS Lui. 2016. Diversified temporal subgraph pattern mining. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2016), San Francisco, CA, USA, 1965–1974*.
- [26] Zhengwei Yang, Ada Wai-Chee Fu, and Ruifeng Liu. 2016. Diversified Top-k Subgraph Querying in a Large Graph. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 1167–1182.

Scalable and Dynamic Regeneration of Big Data Volumes

Anupam Sanghi, Raghav Sood, Jayant Haritsa
 Indian Institute of Science
 Bangalore, India
 {anupam,raghav,haritsa}@dsl.cds.iisc.ac.in

Srikanta Tirthapura
 Iowa State University
 Ames, USA
 snt@iastate.edu

ABSTRACT

A core requirement of database engine testing is the ability to create synthetic versions of the customer’s data warehouse at the vendor site. A rich body of work exists on synthetic database regeneration, but suffers critical limitations with regard to: (a) maintaining statistical fidelity to the client’s query processing, and/or (b) scaling to large data volumes. In this paper, we present **HYDRA**, a workload-dependent database regenerator that leverages a declarative approach to data regeneration to assure volumetric similarity, a crucial aspect of statistical fidelity, and materially improves on the prior art by adding scale, dynamism and functionality. Specifically, Hydra uses an optimized linear programming (LP) formulation based on a novel *region-partitioning* approach. This spatial strategy drastically reduces the LP complexity, enabling it to handle query workloads on which contemporary techniques fail. Second, Hydra incorporates deterministic post-LP processing algorithms that provide high efficiency and improved accuracy. Third, Hydra introduces the concept of *dynamic regeneration* by constructing a minuscule *database summary* that can on-the-fly regenerate databases of arbitrary size during query execution, while obeying volumetric specifications derived from the query workload. A detailed experimental evaluation on standard OLAP benchmarks demonstrates that Hydra can efficiently and dynamically regenerate large warehouses that accurately mimic the desired statistical characteristics.

1 INTRODUCTION

In industrial practice, a common requirement for database vendors is to adequately test their database engines with representative data and workloads that accurately mimic the data processing environments at customer deployments. This need can arise either in the analysis of problems currently being faced by clients, or in proactively assessing the performance impacts of planned engine upgrades on client applications. While, in principle, clients could transfer their original data and workloads to the vendor for the intended evaluation purposes, this is often infeasible due to privacy and liability concerns. Moreover, even if a client is willing to share the data, transferring and storing the data at the vendor’s site may prove to have impractical space and time overheads, especially in the anticipated Big Data era. For instance, if a customer faces a problem on exabyte (10^{18}) sized relational tables, transferring and storing such data is likely to be infeasible even on the best of systems. Therefore, an important requirement, looking into the future, is to be able to *dynamically* regenerate representative databases, at query execution time that accurately mimic the behavior of the client’s data processing environment.

A rich body of literature exists on data regeneration, beginning with *workload-independent* techniques (e.g [12, 15]), which provide scalable and efficient solutions, but fail to retain complex statistical characteristics such as the sizes of intermediate relations created during execution of a query plan. To address this problem, a particularly potent approach of *workload-dependent* database regeneration was introduced in QAGen [11], and has served as the foundation for many of the practicable systems proposed over the last decade [6, 18]. Workload-dependent techniques aim to generate synthetic data whose behavior is *volumetrically similar* to the client database on the pre-specified query workload. That is, assuming a common choice of query execution plans at the client and vendor sites (ensured through “plan forcing” [3] or “metadata matching” [8]), the output row cardinalities of individual operators in these plans are very similar in the original and synthetic databases. This similarity helps to preserve the multi-dimensional layout and flow of the data, a pre-requisite for achieving similar performance on the client’s workload. As a case in point, the DataSynth [6, 7] tool from Microsoft expresses such volumetric constraints as a Linear Program (LP) whose solution is used to construct the synthetic database.

A common limitation of contemporary techniques (reviewed in detail in Section 8), is that they run into issues of *scale* and *efficiency* at one stage or the other in the regeneration pipeline. This is partly due to their focus on *materialized* static solutions, making them impractical at large volumes. Further, the ability to scale to large query workloads and data volumes has not been clearly established, and validations have been typically restricted to relatively simple and small benchmarks such as TPC-H [2]. These limitations become especially problematic from a futuristic “Big Data” perspective, where we have to contend with enormous data volumes and complex query workloads.

To materially address this challenge, we present **HYDRA**, a data regeneration tool, which ensures that scale and efficiency are addressed through *the entire regeneration pipeline*. As a concrete example, Hydra was able to accurately regenerate the data processing environment of a 100 GB TPC-DS client database with a workload of 131 distinct representative queries, by generating a *database summary* in less than 2 minutes on a vanilla machine. This summary can be used to statically generate a materialized database, or more potently, to *dynamically* regenerate the desired database during query execution. When the former option is chosen, the static database was successfully created in less than 11 minutes. It is important to note here that the summary construction time is *independent* of the data scale – therefore, even the exabyte-sized data scenario alluded to earlier could be modeled in just a few minutes using Hydra!

The key contributions of Hydra are the following:

Extended Workload Coverage: Hydra incorporates a novel LP formulation technique, *region-partitioning*, that can encode volumetric constraints with an LP of low complexity. When compared with the *grid-partitioning* approach used in DataSynth, region-partitioning reduces the LP complexity by many orders of magnitude. For instance, an LP with

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

more than a *billion* variables in DataSynth is reduced to an LP with a few *thousand* variables in Hydra – in fact, in this case, the LP solver crashes on the DataSynth formulation, but runs to completion in less than a minute on the Hydra formulation. The beneficial outcome of the low LP complexity is that it facilitates the efficient handling of much richer query workloads.

Apart from enhancing the workload scale, Hydra also expands the database scope to include relational schemas that have DAG-structured dependency graphs, and the query scope to include DNF filter predicates.

Database Summary and Dynamic Regeneration: A unique feature of our data regeneration approach is that it delivers a *database summary* as the output, rather than the static data itself. This summary is of negligible size, depending only on the query workload and *not* on the database scale. It can be used for *dynamically* generating data during query execution, or for materializing static relations if so desired. This summary-based approach eliminates the enormous time and space overheads incurred by prior techniques in generating and storing data before initiating analysis.

Accuracy with Efficiency: Hydra replaces the *sampling-based* approach to data regeneration in DataSynth by a *deterministic alignment* strategy. The alignment operates directly on the database summary, and is therefore extremely efficient. Further, it does not suffer the probabilistic errors that affect the sampling approach, and therefore delivers better fidelity with regard to volumetric similarity.

Enhanced Evaluation: We evaluate Hydra on a diverse workload of 100-plus queries constructed from the complex TPC-DS benchmark, and the results show that it can efficiently regenerate databases for such workloads at various data scales. Further, our evaluation is more comprehensive than prior techniques, which have largely been evaluated on simpler and small-sized query workloads operating on modest databases. For instance, DataSynth has been evaluated on simple TPC-H database environments that resulted, with their formulation, in LPs with only a few thousand variables.

Integration with CODD: CODD [8] is a graphical tool through which database environments with desired meta-data characteristics can be efficiently simulated without persistently generating and/or storing their contents – i.e. a “dataless” approach. We have integrated Hydra with CODD, thus providing an end-to-end system that fully replicates the client data processing environment at the vendor’s site, and is compliant with the CODD’s “dataless” philosophy.

Organization. The remainder of this paper is organized as follows: A brief background on the key underlying concepts is outlined in Section 2. The Hydra architecture is presented in Section 3, and our new region-based LP formulation in Section 4. The database summary generator and the tuple generator are described in Sections 5 and 6, respectively. Our experimental results are analyzed in Section 7. Related work is reviewed in Section 8, and our conclusions are summarized in Section 9.

2 PRELIMINARIES

In this section, we provide background information on the key foundations – Annotated Query Plans [11] and Cardinality Constraints [6] – that lie under this data regeneration framework.

2.1 Annotated Query Plans

Consider a toy scenario (for ease of presentation) where the client has the database schema shown in Figure 1a, where pk and fk refer to primary-key and foreign-key attributes, respectively.

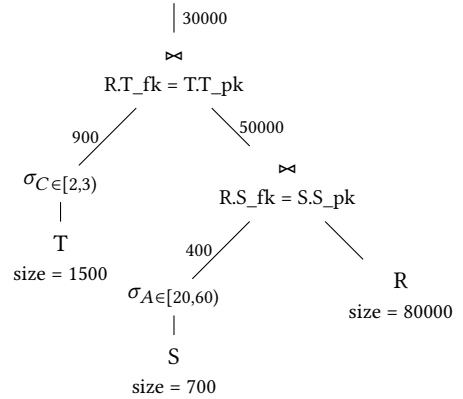
A sample client query on this schema is shown in Figure 1b, with the corresponding query execution plan in Figure 1c. Note that this execution plan has the output edge of each operator annotated with the associated row cardinality (as evaluated during the client’s execution) – for instance, there are 50000 rows resulting from the join of R and (filtered) S. Such a plan is referred to as an “Annotated Query Plan” (AQP) in [11]. The goal now is to generate synthetic data at the vendor site such that when the above query is executed on this data, we obtain an identical, or very similar, AQP.

R (<u>R_pk</u> , S_fk, T_fk)	S (<u>S_pk</u> , A, B)	T (<u>T_pk</u> , C)
-------------------------------	-------------------------	----------------------

(a) Database Schema

select * from R, S, T where R.S_fk = S.S_pk and R.T_fk = T.T_pk and S.A >= 20 and S.A < 60 and T.C >= 2 and T.C < 3

(b) Example Query



(c) Annotated Query Plan (AQP)

$ R = 80000$	$ S = 700$	$ T = 1500$
$ \sigma_{S.A \in [20,60]}(S) = 400$	$ \sigma_{T.C \in [2,3]}(T) = 900$	
$ \sigma_{S.A \in [20,60]}(R \bowtie S) = 50000$		
$ \sigma_{S.A \in [20,60]} \wedge T.C \in [2,3]}(R \bowtie S \bowtie T) = 30000$		

(d) Cardinality Constraints (CCs)

Figure 1: Example Database Scenario

2.2 Cardinality Constraints

A unified and declarative mechanism for representing AQP data characteristics, called *cardinality constraints* (CCs), was proposed in [6]. For instance, the CCs expressing the AQP of Figure 1c are shown in Figure 1d. The data regeneration technique takes the schematic information and the set of CCs from the client site and produces synthetic data that closely meets these CCs. To make the problem tractable, it is assumed that CCs consist of filters on only *non-key* attributes, and that all joins are between primary keys and foreign keys, typically the case in data warehouses.

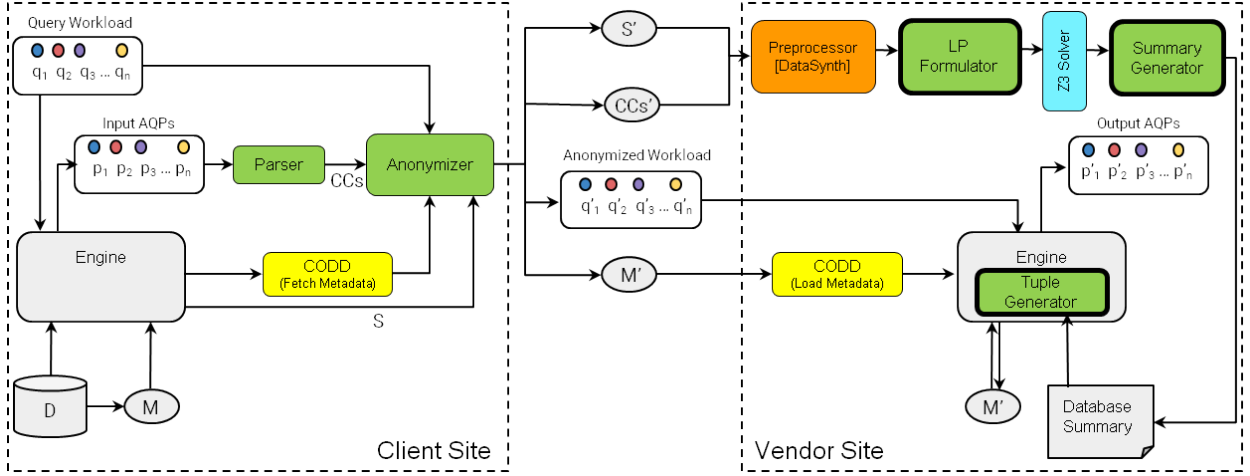


Figure 2: Hydra Architecture

3 THE HYDRA ARCHITECTURE

In this section, we present an overview of Hydra’s architecture, along with a summary of its various components and their interactions with the database engine. A pictorial view of the architecture is presented in Figure 2 – in this picture, the green boxes represent the new components designed specifically for Hydra. Among these, the primary components are the LP Formulator, the Summary Generator, and the Tuple Generator, all shown with thick borders. The other modules have been sourced from the literature, including the preprocessor (orange) from DataSynth [7], the Codd metadata processor (yellow) [8], and the Z3 solver (blue) [14]. (Refer [21] for complete details.)

3.1 Client Site

The information flow from the client to the vendor is as follows: At the client site, Hydra fetches the schema information (S), and the query workload ($q_1, q_2, q_3, \dots, q_n$) with its corresponding AQPs ($p_1, p_2, p_3, \dots, p_n$) obtained from the database engine. The AQPs are converted to equivalent cardinality constraints (CCs) using a Parser. The metadata (M) from the database catalogs is captured with the help of Codd. In order to address client security concerns, all this information (schema, metadata, queries and CCs) is passed through an Anonymizer that suitably masks the information before shipping it to the vendor. Also in this process, non-numeric constants appearing in the queries and plans are mapped to numbers to facilitate LP formulation at the vendor site. Due to this mapping, the final database summary generated at the vendor site also consists of only numeric datatypes. It is possible to reverse this mapping to get back the original datatypes, but is not a relevant consideration with regard to satisfying CCs .

3.2 Vendor Site

The main modules at the vendor site are as follows:

Preprocessor [7]: In this module, sourced from DataSynth, the schema information and CCs obtained from the client are processed to create the input for the LP Formulator. Each relation is solved independently, and this process is initiated by first creating a *view* comprised of its own non-key attributes, augmented with the non-key attributes of the relations on which it depends through referential constraints (both directly or transitively). This transformation results in replacing the join-expression present in

a CC with a view that covers all the attributes (non-key) featured in the relations participating in the join-expression. As a case in point, following views are generated for the example in Figure 1:

$$R_view(A, B, C) \quad S_view(A, B) \quad T_view(C)$$

Further, the last two constraints in Figure 1d can be rewritten as:

$$\begin{aligned} |\sigma_{A \in [20, 60]}(R_view)| &= 50000 \\ |\sigma_{A \in [20, 60] \wedge C \in [2, 3]}(R_view)| &= 30000 \end{aligned}$$

An LP is independently formulated for each view created by the above process. Since the LP complexity is adversely affected by the number of attributes in the view, the view is first decomposed into a set of *sub-views* to reduce the effective complexity. This is achieved as follows: Construct a “view-graph” by first creating a node for each attribute, and then inserting an edge between a pair of nodes if the corresponding attributes appear together in one or more CCs . Further, additional edges are added (if required) to make the view-graph to be *chordal*, a property required to ensure acyclicity in the subsequent processing. Now, the sub-views are identified as the *maximal cliques* in the view-graph.

LP Formulator and Solver: For each view, the LP Formulator takes as input the corresponding set of subviews and applicable CCs , and then constructs the LP. The domain corresponding to each sub-view is partitioned into regions using a novel *region-partitioning* algorithm that takes as input the different cardinality constraints. There is one variable for each region, corresponding to the number of tuples chosen from the region. Each cardinality constraint is encoded as an LP constraint on these variables, and the solution of the LP is used in deciding which tuples to include in the sub-view. The complete details of this algorithm are enumerated in Section 4.

Our region-partitioning strategy is in marked contrast to the *grid-partitioning* strategy used in DataSynth. Grid-Partitioning first intervalizes the domain of each attribute based on the constants appearing in the CCs , and divides the domain into a grid aligned with the interval boundaries for each attribute. If a sub-view has n attributes, and each attribute gets divided into ℓ intervals, then the domain of the sub-view is partitioned into a grid of ℓ^n cells. For each cell in the grid, a variable is created that represents the number of data rows present in that cell. In contrast, our region-partitioning strategy divides the domain into only the number of regions required to precisely write out each

cardinality constraint, and assigns one variable to each region – this typically leads to far fewer variables than grid-partitioning.

To make the above concrete, consider a single view “Person” with the following three selection CCs:

$$\begin{aligned} |\text{age} < 40 \wedge \text{salary} < 40\text{K}(\text{Person})| &= 1000 \\ |20 \leq \text{age} < 60 \wedge 20\text{K} \leq \text{salary} < 60\text{K}(\text{Person})| &= 2000 \\ |\text{Person}| &= 8000 \end{aligned}$$

Grid-partitioning divides the domain of the view as shown in Figure 3a. With a variable assigned to each grid cell, there is a total of 16 variables. In contrast, the region-partitioning strategy partitions the space into 4 regions as shown in Figure 3b, resulting in a tally of only 4 variables.

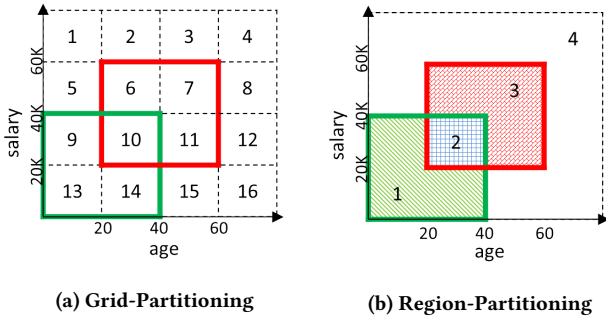


Figure 3: Grid-Partitioning vs Region-Partitioning

The CCs of Person, expressed in terms of LP constraints, are shown below in Figure 4a and 4b for grid-partitioning and region-partitioning, respectively.

$$\begin{aligned} x_9 + x_{10} + x_{13} + x_{14} &= 1000 & y_1 + y_2 &= 1000 \\ x_6 + x_7 + x_{10} + x_{11} &= 2000 & y_2 + y_3 &= 2000 \\ x_1 + x_2 + \dots + x_{16} &= 8000 & y_1 + y_2 + y_3 + y_4 &= 8000 \end{aligned}$$

(a) Grid-Partitioning (b) Region-Partitioning

Figure 4: LP Constraints

The LPs are passed on to the solver, which provides one of the feasible solutions as the output – we have used Z3 [14], a popular SMT solver, to implement this functionality. With region-partitioning, the LP is usually much simpler due to the smaller number of variables. Further, as the cardinality constraints get more complex, the differences in complexity of the LPs produced by region-partitioning and grid-partitioning become more pronounced. This effect is quantified in Section 7.

Summary Generator: This module generates the database summary from the LP solutions obtained on the views. Since partitioning is carried out at a sub-view level, the LP solution, which is expressed in terms of sub-view variables, needs to be mapped to equivalents in the original view space. A sampling-based approach was proposed in [6] for this purpose – for example, say a view (A, B, C) is split into a pair of sub-views (A, B) and (B, C) , the algorithm computes the distributions $Prob(A, B)$ and $Prob(C|B)$. Then, each tuple is generated by first sampling a point from the former distribution, and then sampling a point from the latter conditioned on this outcome.

However, we have chosen not to take this approach since the computational overheads incurred are enormous, and the sampling process introduces errors in volumetric fidelity. Instead, we have designed and implemented an alternative data-scale free, deterministic alignment algorithm (details in Section 5), which produces an intermediate database summary in the output. This component is also responsible for ensuring that the generated summary obeys referential integrity. Finally, summarized relations from corresponding view summary are obtained. An example database summary finally obtained from the AQP shown in Figure 1c, along with additional two AQPs, is shown in Figure 5. Here, entries of the type $a - b$ in the primary key columns (e.g. 101-250 for S_pk in table S), mean that the relation has $b - a + 1$ tuples with values $(a, a + 1, a + 2, \dots, b)$ for that column, keeping the other columns unchanged.

R			S			T	
R_pk	S_fk	T_fk	S_pk	A	B	T_pk	C
1 - 30K	321	1	1-100	0	15	1-600	0
30001 - 50K	621	601	101-250	20	15	601-1500	2
50001 - 60K	71	601	251-500	20	10		
60001 - 70K	121	1	501-700	0	5		
70001 - 80K	1	1					

Figure 5: Example Database Summary

Tuple Generator: The Tuple Generator resides in the database engine. It ensures that whenever a query is fired, data is not fetched from the disk but instead gets generated *on-demand*, using the database summary. The details of this component and its implementation in PostgreSQL are presented in Section 6.

We note in closing that in order to ensure the execution plan chosen at the vendor site is the same as that in the client site, metadata matching is implemented in Hydra using CODD’s metadata transfer feature.

4 LP FORMULATION

An LP for a view V is constructed as follows: For each sub-view s in V , every CC that is within its scope is formulated as an LP constraint. Since sub-views may share common attributes, additional *consistency constraints* are added to the LP to ensure that the marginal distributions along the common set of attributes are identical in the solutions for the sub-views.

In this section, we first present the mathematical basis underlying our formulation of LP constraints for a set of CCs applicable on a sub-view. We then present an algorithm that partitions the domain into the minimum number of regions required to capture each CC precisely, resulting in an LP with the optimal number of variables. Finally, we discuss the formulation of additional consistency constraints to ensure consistency across multiple sub-views belonging to V .

4.1 Mathematical Basis for LP Formulation

Let n denote the number of attributes in the given sub-view s , \mathcal{D}_i the domain of the i th attribute, and \mathcal{D} the data universe $\mathcal{D}_1 \times \mathcal{D}_2 \times \dots \times \mathcal{D}_n$.

We are given a set of m CCs that are applicable on s . For $1 \leq j \leq m$, each constraint C_j is a pair $\langle \sigma_j, k_j \rangle$ where σ_j is a selection predicate and k_j is a non-negative integer equal to the number of rows satisfying predicate σ_j . We assume that each predicate is in disjunctive normal form (DNF).

Simple LP Formulation. Let us first consider a simple way of formulating an LP that encodes all CCs. For each tuple $t \in \mathcal{D}$, assign a variable x_t that denotes the number of copies of t in the sub-view s . Then, the LP formulation shown in Figure 6 ensures that a feasible solution satisfies all CCs, including a constraint on the total size of s .

The problem with this formulation is that the number of variables in the resulting LP is as large as the size of the universe \mathcal{D} . Hence, it is infeasible to work directly with this formulation.

$$\begin{aligned}
(1) & \text{ For each } t \in \mathcal{D}, x_t \geq 0 \\
(2) & \left[\sum_{t \in \mathcal{D}} x_t \right] = k \\
(3) & \text{ For each } j, 1 \leq j \leq m, \left[\sum_{t: \sigma_j(t)=\text{true}} x_t \right] = k_j
\end{aligned}$$

Figure 6: Simple LP formulation

Reduced LP Formulation. We can derive an LP with far fewer variables as follows: We first note that in the simple formulation, variables corresponding to a pair of points $t_1, t_2 \in \mathcal{D}$ that behave identically with respect to a constraint C_j (i.e. $\sigma_j(t_1) = \sigma_j(t_2)$) can be combined together as $(x_{t_1} + x_{t_2})$, for the purposes of satisfying constraint C_j . If this is true that with respect to every constraint C_j for $j = 1 \dots m$, $\sigma_j(t_1) = \sigma_j(t_2)$, then there is no need to treat t_1 and t_2 separately – instead, they can be combined into a single region, and the variables x_{t_1} and x_{t_2} can be merged into a single variable $(x_{t_1} + x_{t_2})$ in every equation, leading to fewer variables in the LP. By repeating this variable merging process recursively until it is no further possible, we arrive at a vastly reduced LP.

We hasten to add that the above LP construction process based on merging variables is only for illustrating the concept – the actual algorithm employed in our system *directly* derives the regions, as described in Section 4.2.

For constraint C and $t \in \mathcal{D}$, let $C(t)$ be an indicator variable:

$$C(t) = \begin{cases} \text{true} & \text{if } t \text{ satisfies } C \\ \text{false} & \text{otherwise} \end{cases}$$

Definition 4.1. For a pair of points $p, q \in \mathcal{D}$ and a set of constraints \mathbb{C} , we say $pR^{\mathbb{C}}q$ if for each $C \in \mathbb{C}$, $C(p) = C(q)$.

OBSERVATION 1. $R^{\mathbb{C}}$ is an equivalence relation on \mathcal{D} .

PROOF. It can be easily seen that $R^{\mathbb{C}}$ is reflexive and symmetric. For transitivity, suppose that for $p, q, r \in \mathcal{D}$, $pR^{\mathbb{C}}q$ and $qR^{\mathbb{C}}r$. Note that for each $C \in \mathbb{C}$, it must be true that $C(p) = C(q)$ and $C(q) = C(r)$. Therefore, it must be true that $C(p) = C(r)$ for each $C \in \mathbb{C}$, showing that the relation is transitive. \square

A partition of \mathcal{D} is a set of subsets of \mathcal{D} such that every element $x \in \mathcal{D}$ is in exactly one of these subsets. The individual sets in a partition are called *blocks*.

Definition 4.2. A set of points b is said to be valid with respect to a set of constraints \mathbb{C} if for any two points $p, q \in b$, $pR^{\mathbb{C}}q$. Given a set of constraints \mathbb{C} , a partition \mathbb{P} of \mathcal{D} is said to be a *valid partition* if for each block $b \in \mathbb{P}$, b is valid with respect to \mathbb{C} .

In a valid partition of \mathcal{D} with respect to \mathbb{C} , any pair of points within the same block satisfy the same set of CCs. Once we

obtain a valid partition \mathbb{P} , the LP can be re-formulated as shown in Figure 7. Instead of a variable for each point $t \in \mathcal{D}$, there is now a single variable x_b for each block $b \in \mathbb{P}$ representing the number of tuples of the sub-view that are contained in this block. Note that the tuples in a sub-view need not be unique, therefore x_b may include duplicates in its count.

$$\begin{aligned}
(1) & \text{ For each } b \in \mathbb{P}, x_b \geq 0 \\
(2) & \left[\sum_{b \in \mathbb{P}} x_b \right] = k \\
(3) & \text{ For each } j, 1 \leq j \leq m, \left[\sum_{b: \sigma_j(b)=\text{true}} x_b \right] = k_j
\end{aligned}$$

Figure 7: Reduced LP formulation

The total number of variables in the reduced LP shown in Figure 7 is equal to the number of blocks in the partition \mathbb{P} and is potentially much smaller than the number of variables in the original LP, shown in Figure 6. Since we desire an LP with the smallest number of variables, we look for a valid partition of \mathcal{D} with the minimum number of blocks. A valid partition with respect to \mathbb{C} is an *optimal partition* if it has the smallest number of blocks from among all valid partitions of \mathcal{D} with respect to \mathbb{C} .

LEMMA 4.3. *The quotient set of \mathcal{D} by $R^{\mathbb{C}}$ is the (unique) optimal partition of \mathcal{D} with respect to \mathbb{C} .*

PROOF. Let \mathbb{P}_1 denote the quotient set¹ of \mathcal{D} by $R^{\mathbb{C}}$. By the definition of an equivalence relation, for any block $b \in \mathbb{P}_1$, all points in b are related to each other by $R^{\mathbb{C}}$, and hence \mathbb{P}_1 is a valid partition.

Suppose that \mathbb{P}_1 is not the unique optimal partition. Then, there must exist another valid partition \mathbb{P}_2 such that $\mathbb{P}_2 \neq \mathbb{P}_1$ and $|\mathbb{P}_2| \leq |\mathbb{P}_1|$. This implies that there exist two points $p, q \in \mathcal{D}$ such that p and q are in different blocks in \mathbb{P}_1 , but in the same block in \mathbb{P}_2 . Since p and q belong to different blocks in \mathbb{P}_1 , it must be true that p and q are not related by $R^{\mathbb{C}}$. But, in \mathbb{P}_2 points p and q belong to the same block, which implies that \mathbb{P}_2 cannot be a valid partition, a contradiction. \square

4.2 Deriving the Optimal Partition

We now present an algorithm to derive the optimal partition of \mathcal{D} with respect to \mathbb{C} . Each constraint $C \in \mathbb{C}$ is in DNF, and is expressed as the union of many smaller “sub-constraints”. Each sub-constraint is the conjunction of many per-attribute constraints, and each per-attribute constraint is a constraint on the values that the attribute is permitted to take. For example, the following constraint on attributes A_1 and A_2 :

$$((A_1 \leq 20) \wedge (A_2 > 30)) \vee (A_1 > 50)$$

is divided into the basic sub-constraints:

$$(A_1 \leq 20) \wedge (A_2 > 30) \text{ and } (A_1 > 50)$$

Algorithm 1 (Optimal Partition) takes a set of DNF constraints as input, and returns a partition with the smallest number of regions with respect to this set. Internally, it invokes Algorithm 2 (Valid Partition) that takes a set of sub-constraints as input and returns a valid partition of the domain with respect to this set.

¹The quotient set is the set of equivalence classes resulting from $R^{\mathbb{C}}$ on \mathcal{D} .

Algorithm 1: Optimal Partition(\mathcal{D}, \mathbb{C})

Input: Universe \mathcal{D} , set of DNF constraints \mathbb{C} **Output:** An optimal partition \mathbb{P}^* of \mathcal{D} subject to \mathbb{C}

- 1 Generate the set of sub-constraints \mathbb{C}' resulting from the constraints in \mathbb{C} ;
 - 2 Construct a valid partition \mathbb{P}' of \mathcal{D} subject to \mathbb{C}' using Valid-Partition(\mathcal{D}, \mathbb{C}') (Algorithm 2);
 - 3 For each block $b \in \mathbb{P}'$, compute the label $\ell(b)$, equal to the set of all constraints in \mathbb{C} that b satisfies. Let L denote the set of all distinct labels from $\{\ell(b) | b \in \mathbb{P}'\}$;
 - 4 Coarsen partition \mathbb{P}' into \mathbb{P}^* as follows: For each label $l \in L$, merge all blocks in \mathbb{P}' whose labels equal l into a single block;
 - 5 Return \mathbb{P}^* ;
-

LEMMA 4.4. Given a set of DNF constraints \mathbb{C} , Algorithm 1 returns an optimal partition of \mathcal{D} with respect to \mathbb{C} .

PROOF. As in the algorithm, let \mathbb{C}' denote the set of sub-constraints resulting from constraints in \mathbb{C} . From Lemma 4.7, we know that \mathbb{P}' is a valid partition with respect to \mathbb{C}' . Consider any block $b \in \mathbb{P}'$. Since b is valid with respect to \mathbb{C}' , and each constraint in \mathbb{C}' is stricter than a corresponding constraint in \mathbb{C} , b is valid with respect to \mathbb{C} . Hence, \mathbb{P}' is a valid partition with respect to \mathbb{C} .

Next, consider that each block b^* in \mathbb{P}^* was obtained by merging blocks in \mathbb{P}' that have the same label. For any pair of points p, q in b^* , it is true they satisfy the same set of constraints in \mathbb{C} , showing that \mathbb{P}^* is a valid partition wrt \mathbb{C} . Also, any two blocks in \mathbb{P}^* have distinct labels (if they had the same label, they would have been merged). Therefore, we conclude using arguments similar to Lemma 4.3 that \mathbb{P}^* is an optimal partition of \mathcal{D} with respect to \mathbb{C} . \square

Deriving a Valid Partition for a Set of Sub-Constraints: We now present an algorithm for deriving a valid partition with a small number of blocks, for a set of sub-constraints \mathbb{C} .

Definition 4.5. For a sub-constraint C and dimension i , let C^i denote the restriction (projection) of C to dimension i . Further, let $C_1^i = \bigwedge_{k=1 \dots i} C^k$ denote the restriction of C to dimensions $1, 2, \dots, i$. For instance, if $C = (A_1 \geq 1) \wedge (A_2 \geq 4) \wedge (A_2 \leq 5) \wedge (A_3 > 6)$, then $C^2 = (A_2 \geq 4) \wedge (A_2 \leq 5)$, and $C_1^2 = (A_1 \geq 1) \wedge (A_2 \geq 4) \wedge (A_2 \leq 5)$. For convenience, if C does not have a constraint along dimension i , then C^i is defined to be “true”.

Our algorithm, described in Algorithm 2, proceeds iteratively, one dimension at a time. Before processing dimension i , it has a partition of \mathcal{D} that is a valid partition subject to constraints along dimensions 1 till $(i-1)$. In processing dimension i , it refines the current partition as follows: For each block b in the current partition, it appropriately divides the block along dimension i if there is a constraint $C \in \mathbb{C}$ such that there are some points in b that satisfy constraint C^i , and some that do not.

Definition 4.6. A constraint C is said to split a block $b \subseteq \mathcal{D}$ if there exist a pair of points $p_1, p_2 \in b$ such that $C(p_1) = \text{true}$ and $C(p_2) = \text{false}$. If C splits b , then refining b by C partitions b into two subsets $b^+(C) = \{x \in b | C(x) = \text{true}\}$ and $b^-(C) = \{x \in b | C(x) = \text{false}\}$.

LEMMA 4.7. Given a set of sub-constraints \mathbb{C} , Algorithm 2 returns a valid partition of \mathcal{D} with respect to \mathbb{C} .

Algorithm 2: Valid-Partition(\mathcal{D}, \mathbb{C})

Input: Universe \mathcal{D} , set of sub-constraints \mathbb{C} **Output:** A valid partition \mathbb{P} of \mathcal{D} subject to set of sub-constraints \mathbb{C}

- 1 $\mathbb{P}^0 = \{\mathcal{D}\}$ // A partition with one set, \mathcal{D} .
 - 2 **for** i from 1 to n **do**
 - 3 $M \leftarrow \mathbb{P}^{i-1}$;
 - 4 **foreach** $C \in \mathbb{C}$ **do**
 - 5 $M' \leftarrow \emptyset$;
 - 6 **foreach** block $b \in M$ **do**
 - 7 **if** C^i splits b **then**
 - 8 Let b^+ and b^- result from refining b with C^i ;
 - 9 Add b^+ and b^- to M' ;
 - 10 **else**
 - 11 Add b to M' ;
 - 12 $M \leftarrow M'$;
 - 13 $\mathbb{P}^i \leftarrow M$;
 - 14 **Return** \mathbb{P}^n ;
-

PROOF. For $1 \leq i \leq n$, let $\mathbb{C}_1^i = \{C_1^i | C \in \mathbb{C}\}$. We show by induction on i that after the i th iteration of the outermost for loop in the algorithm, \mathbb{P}^i contains a valid partition of \mathcal{D} with respect to \mathbb{C}_1^i . Since $\mathbb{C}_1^n = \mathbb{C}$, it follows that after n iterations, \mathbb{P}^n contains a valid partition of \mathcal{D} with respect to \mathbb{C} . We consider $i = 0$ as the base case, and the set \mathbb{C}_1^0 as a set of “always true” constraints. Hence, \mathbb{P}^0 , which consists of only one element, \mathcal{D} , is a valid partition with respect to \mathbb{C}_1^0 .

For the inductive step, suppose that for $i > 0$, \mathbb{P}^{i-1} is a valid partition of \mathcal{D} with respect to \mathbb{C}_1^{i-1} . For each block $b \in \mathbb{P}^{i-1}$, two cases are possible: (1) b is not split by C^i , for any $C \in \mathbb{C}$. Then b is valid with respect to \mathbb{C}_1^i , and will be retained in \mathbb{P}^i . (2) b is split by one more constraints C^i . The algorithm iterates through all such constraints that split b , and partitions block b such that every resulting block is valid with respect to each C^i , $C \in \mathbb{C}$.

We next note that \mathbb{P}^i is indeed a partition of \mathcal{D} (i.e. the union of all blocks equals \mathcal{D}). To see this observe that each block $b \in \mathbb{P}^{i-1}$ is either present in \mathbb{P}^i or has been refined and all its constituent blocks (whose union equals b) are in \mathbb{P}^i . Thus, \mathbb{P}^i is a valid partition with respect to \mathbb{C}_1^i . This proves the inductive step. \square

Consistency Constraints. Since different sub-views can have common attribute(s), additional constraints need to be added to ensure that their distributions for the common attribute(s) are the same. In order to do so, we may need to further refine the partition generated from the above procedure. Specifically, consider a pair of sub-views s_1 and s_2 with attribute sets \mathbb{A}_1 and \mathbb{A}_2 respectively, such that $\mathbb{A}_1 \cap \mathbb{A}_2 \neq \emptyset$. Let $\mathcal{D}^1 = \prod_{i \in \mathbb{A}_1} \mathcal{D}_i$, and $\mathcal{D}^2 = \prod_{j \in \mathbb{A}_2} \mathcal{D}_j$ be the corresponding domains for s_1 and s_2 respectively, and $\mathcal{D}^{1,2} = \prod_{i \in \mathbb{A}_1 \cap \mathbb{A}_2} \mathcal{D}_i$. Let the partitions obtained on \mathcal{D}^1 and \mathcal{D}^2 be \mathbb{P}_1 and \mathbb{P}_2 , respectively. In order to keep \mathbb{P}_1 and \mathbb{P}_2 consistent with each other, we need to ensure that their region boundaries are aligned with each other, and this is achieved by refining \mathbb{P}_1 and \mathbb{P}_2 so that they have common boundaries along dimensions $\mathbb{A}_1 \cap \mathbb{A}_2$. We consider the union of the “split points” of \mathbb{P}_1 and \mathbb{P}_2 along dimensions $\mathbb{A}_1 \cap \mathbb{A}_2$ and further for each block in \mathbb{P}_1 (and \mathbb{P}_2), we refine this block until it no longer crosses such a split point. Finally, we add LP constraints that equate distributions of the common attributes in \mathbb{P}_1 and \mathbb{P}_2 .

5 DATABASE SUMMARY GENERATOR

This component takes the LP solution for each view as the input and generates the database summary, which as mentioned previously, can be used for dynamically generating data for query execution, or can optionally be used to generate the materialized database.

Recall that a variable in the LP (for a view) represents an underlying block in a sub-view’s partition, and its assigned value is the number of rows present in that block – this value is hereafter referred to generically as NUMTUPLES. The collection of NUMTUPLES values represent the sub-view solutions, and these solutions are integrated to obtain the solution for the complete view. However, since each view is solved independently, the referential constraints that exist between the corresponding relations may be lost in these view solutions. Therefore, they may have to be modified to ensure global consistency. Finally, it is necessary to extract relations from the views in order to populate the database. Accordingly, the summary generator component in Hydra is responsible for the following sequence of tasks:

- (1) Constructing a solution for complete views
- (2) Instantiating view summaries
- (3) Making view summaries consistent wrt each other
- (4) Extracting relation summaries from view summaries

5.1 Constructing Solution for the View

For integrating the sub-view solutions to obtain the collective solution for the complete view, we first *order* the sub-views. Then, we iteratively build the view-solution by *aligning* and *merging* the next sub-view solution in the given order. Let \mathbb{S} denote the input list of sub-view solutions, and *viewSol* be the final view solution that we wish to compute. Algorithm 3 describes the high-level process for constructing *viewSol* from \mathbb{S} , and its ordering, aligning and merging procedures are described in the remainder of this sub-section.

Algorithm 3: View Solution Construction

```

1  $\mathbb{S} \leftarrow \text{ORDERSUBVIEWS}(\mathbb{S});$ 
2  $\text{viewSol} \leftarrow \emptyset;$ 
3 foreach  $s \in \mathbb{S}$  do
4    $\text{viewSol}, s \leftarrow \text{ALIGN}(\text{viewSol}, s);$ 
5    $\text{viewSol} \leftarrow \text{MERGE}(\text{viewSol}, s);$ 

```

5.1.1 Sub-View Ordering. Ordering is implemented through a greedy iterative algorithm where we can start with any sub-view as the first choice. Subsequently, at iteration i , let the set of visited sub-views until now be \mathbb{S} . A sub-view s from outside this set can be chosen to be the next in the ordering only if it satisfies the following condition: On removing the common vertices between s and \mathbb{S} in the (chordal) view-graph, there should not exist any path between the remaining vertices of s and the remaining vertices of \mathbb{S} . This algorithm is described in detail in [21].

5.1.2 Aligning. After obtaining the sub-view merge order as per above, in every iteration we merge the next sub-view solution (s) in the sequence to the current view-solution (*viewSol*), after a process of alignment. The alignment algorithm is a two step exercise, as shown in the example of Figure 8:

Solution Sorting: First, the *viewSol* and s solutions are each sorted on their common set of attributes to facilitate direct

comparison of their matching ranges. For instance, the solutions A, B and A, C in Figure 8a are each sorted on the intervals enumerated in the common attribute A .

Row Splitting: Our addition of consistency constraints during the LP formulation ensured that the distribution of tuples along the common set of attributes is the same in the various sub-views. Therefore it is easy to see that the sum of NUMTUPLES values in any interval of the common attributes is the same for the sub-view solutions under alignment. For example, in Figure 8a, the total number of tuples with $A = [40, 60]$ is 30K in both the A, B and A, C solutions. Likewise, the other entries in column A also have matching total number of tuples across the solutions. The align step *splits* the rows in these solutions such that the corresponding rows in both solutions have the same number of tuples. The sub-view solutions of Figure 8a are shown in Figure 8b after undergoing the alignment process, with both solutions now having identical NUMTUPLES in the corresponding rows.

A	B	NUMTUPLES	A	C	NUMTUPLES
[60, inf)	[0, inf)	30K	[60, inf)	[0, inf)	30K
[40, 60)	[0, 5) U [15, inf)	20K	[40, 60)	[2, 3)	30K
[40, 60)	[5, 15)	10K	[20, 40)	[0, 2) U [3, inf)	20K
[20, 40)	[5, 10)	10K			
[20, 40)	[15, inf)	10K			

(a) Sub-view Solution

A	B	NUMTUPLES	A	C	NUMTUPLES
[60, inf)	[0, inf)	30K	[60, inf)	[0, inf)	30K
[40, 60)	[0, 5) U [15, inf)	20K	[40, 60)	[2, 3)	20K
[40, 60)	[5, 15)	10K	[40, 60)	[2, 3)	10K
[20, 40)	[5, 10)	10K	[20, 40)	[0, 2) U [3, inf)	10K
[20, 40)	[15, inf)	10K	[20, 40)	[0, 2) U [3, inf)	10K

(b) View Alignment

A	B	C	NUMTUPLES
[60, inf)	[0, inf)	[0, inf)	30K
[40, 60)	[0, 5) U [15, inf)	[2, 3)	20K
[40, 60)	[5, 15)	[2, 3)	10K
[20, 40)	[5, 10)	[0, 2) U [3, inf)	10K
[20, 40)	[15, inf)	[0, 2) U [3, inf)	10K

(c) Merged View Solution

Figure 8: Align and Merge Example

5.1.3 Merging. This is the last step in the construction of the view solution. Here we simply merge the two solutions obtained after alignment through a “position” based join, where the physically corresponding rows in each solution are combined, with the common attributes being represented once. For example, the aligned solutions of Figure 8b are merge-joined using the positions (or row identifiers) to deliver the final view solution of Figure 8c.

As discussed earlier, DataSynth adopted a sampling algorithm for constructing the view solutions post LP solving. In marked contrast, Hydra *deterministically* generates the view solutions, facilitating us to operate purely in the summary space. There are

two tangible benefits of this deterministic strategy: (a) elimination of the time and space overheads due to sampling, and (b) elimination of sampling-based errors in satisfying CCs.

5.2 Instantiating View Summaries

As shown in Figure 8c, each row in the view solution is comprised of a series of intervals (across various attributes) and the number of tuples in the region represented by these intervals. We now need to decide as to how these tuples are distributed *within* the attribute intervals. Our current solution is very simple: Assign the *entire* cardinality to the *left boundaries* of the intervals. For example, the third row in Figure 8c would result in generation of 10000 tuples all having $A = 40, B = 5, C = 2$ values.

Note that, in principle, we could have used a more sophisticated cardinality distribution within the intervals. However, our simple deterministic choice helps to reduce the subsequent additive errors that are incurred while ensuring referential integrity across views (described in next subsection). This is so because choosing values deterministically within a bucket minimizes the likelihood of encountering an fk value that is not present in the corresponding pk column.

5.3 Making View Summaries Consistent

Since the solution for each view is obtained independently, there could be inconsistencies across them. For example, referring back to the view schema shown in Section 3.2, R_{view} has attributes borrowed from S_{view} and T_{view} , and its solution may feature values that are not present in the corresponding attributes of these two views. To address this problem, we first carry out a *topological sort* on the “referential dependency graph”² and then iteratively make the current view consistent with its predecessors. Since a topological sort is employed, Hydra can handle dependency graphs that are DAGs unlike DataSynth which is restricted to tree traversals.

To make a pair of views V_i and V_j consistent with each other, where V_i is dependent on V_j , we iterate over the rows in the view solution of V_i and look for the value combination that each row has for the attributes borrowed from V_j . If that value combination is not present in the solution of V_j , we add a new row in its solution with the corresponding NUMTUPLES attribute set to 1. This results in an additive error in the total number of tuples in the view as compared to the original AQP at the client. But we hasten to add that the error is a fixed number of rows, determined by the nature of the constraints and the LP solution, and *not* by the data scale. Therefore, at Big Data volumes, the discrepancy can be expected to be minuscule, and our experiments empirically confirm this expectation.

The inter view consistency component is present in DataSynth as well, but since its view solutions are comprised of complete database instantiations, and not just summaries, the time and space overheads incurred for making the views consistent can be large. Moreover, the additive error in DataSynth is amplified due to its inherent sampling errors. Our experiments also capture this distinction between the errors incurred due to referential constraints in Hydra and DataSynth.

5.4 Constructing Relation Summaries

After constructing consistent solutions across all the views, we next need to obtain the corresponding relation summaries. For

²A graph where each relation is represented by a node and an edge (u, v) is added if relation u is dependent on relation v through a referential constraint.

this, we create a summarized relation schema \tilde{R}_i for each relation R_i . This schema consists of all attributes in R_i except the primary key attribute, and additionally, the NUMTUPLES value for each entry in \tilde{R}_i , as sourced from the view solutions.

For the common attributes between the summarized relation and the corresponding view solution, the value combinations and corresponding NUMTUPLES value are directly borrowed from the solution. What remains are the foreign key attributes. For filling a foreign key attribute fk, we need to first consult the view corresponding to fk’s target relation, say V_j . To fill the fk value in row r of \tilde{R}_i , we extract the value combination in row r of view solution of V_j , and then project the attributes corresponding to V_j – let this be denoted by v . Now, we iterate over the solution set of V_j and compute the cumulative sum of the cardinality entries till v is reached. This sum provides the fk value corresponding to the r th row of \tilde{R}_i , and we thus obtain \tilde{R}_i for each relation R_i .

The set of relation summaries, computed as described above, provides the entire database summary – a sample such summary was previously shown in Figure 5 (for simplicity, the figure shows the PK columns instead of the number of tuples).

Like before, DataSynth again iterates over the complete instantiated (consistent) views to construct the corresponding materialized relations. Obviously, this leads to enormous time and space overheads in contrast to our data-scale independent summary based approach.

6 TUPLE GENERATOR

The Tuple Generator component resides inside the database engine, and needs to be explicitly incorporated in the engine codebase by the vendor. As a proof of concept, we have implemented it for the PostgreSQL v9.3 engine by adding a new feature called *datagen*, which is included as a property for each relation in the database. Whenever this feature is enabled for a relation, the scan operator for that relation is replaced with the dynamic generation operator. As a result, during query execution, the executor does not fetch the data from the disk but is instead supplied by the Tuple Generator in an *on-demand* manner, using the available relation summary.

Each row in the relation summary has a value combination and an associated NUMTUPLES entry. We consider the pk values to be the row numbers of the relation. Therefore, to get the r th tuple of a relation R , the pk is chosen as r and the rest of the attributes come from the relation summary. We iterate over the rows of \tilde{R} and take the cumulative sum of the NUMTUPLES entries until the sum exceeds r . Say the summation crosses the value r in j th row of \tilde{R} . Then the rest of the values of the r th tuple are assigned to be precisely the same as those present in the j th row of \tilde{R} . For example, the 120th row of relation S in Figure 5, would be $\langle 120, 20, 15 \rangle$.

Note that this form of tuple generation is expected to be efficient since the attribute value assignments are deterministic and independent, and these expectations are confirmed in the experiments shown in the following section.

7 EXPERIMENTS

We have implemented the Hydra design, described in the previous sections, in a Java tool running to over 15K lines of code. The popular Z3 [14] solver is leveraged to compute solutions for the LP formulations. In this section, we evaluate Hydra’s empirical performance, using our implementation of DataSynth as the comparative yardstick in the analysis.

Database Environment. The TPC-DS [1] decision-support benchmark database, with a default size of 100GB, is used as the baseline in our experiments. The database is hosted on a PostgreSQL v9.3 engine [4] with the hardware platform being a vanilla HP workstation (3.2 GHz 16 core processor, 32 GB memory, 500 GB SSD hard drive) running Ubuntu Linux 16.04.3.

A complex query workload, WL_c , featuring 131 distinct queries (enumerated in [21]), was created by customizing the 99 queries of the benchmark such that only non-key filter predicates and PK-FK joins were retained, and all nested queries were separated into independent sub-queries³. The AQP’s for these queries were generated on the PostgreSQL query processor, resulting in 351 cardinality constraints. The distribution of the cardinalities for these CCs are shown in Figure 9, with the cardinalities measured on a log-scale. The figure clearly indicates that a wide range of cardinalities are present in the constraints, going from a few tuples to almost a billion.

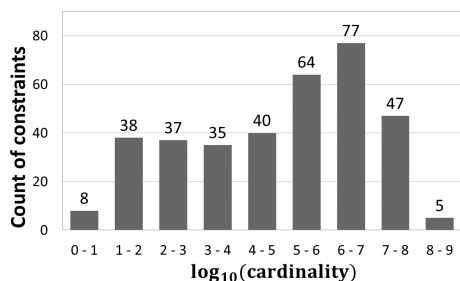


Figure 9: Distribution of Cardinality in CCs (WL_c)

The above constraints result in a large number of geometrically overlapping regions. Hydra, due to its region-partitioning approach, comfortably handles this scenario. In marked contrast, DataSynth, due to its grid-partitioning construction, generates a very large number of LP variables (in the several billion) from the constraints, overwhelming the solver’s capabilities. We therefore also created an alternative simplified query workload, called WL_s , with 311 CCs, wherein the variables created by DataSynth were less than a million, and therefore well within the solver’s processing power.

7.1 Quality of Volumetric Similarity

We begin by investigating how closely the volumetric similarity, with regard to operator output cardinalities, is achieved between the client and vendor sites for the WL_s workload by the Hydra and DataSynth regenerators. This behavior is captured in Figure 10, which plots the percentage of CCs that are within a given relative error of volumetric similarity. From the plot it is evident that Hydra satisfies around 90 percent of the CCs with virtually no error, and the remaining CCs are also satisfied within a relative error of less than 10%. This is in contrast to DataSynth, which accurately satisfies around 80 percent of the CCs, but then incurs as much as 60% relative error to achieve complete coverage of the remaining CCs.

There are two reasons for the error-prone behavior of DataSynth: (1) the probabilistic sampling technique, and (2) the maintenance of referential integrity. While Hydra also is forced to

³Similar to DataSynth, the restriction to non-key-based filters is because the conversion from relations to views lose the key attributes. Likewise, only PK-FK joins are supported since they are inherently present in the design of views.

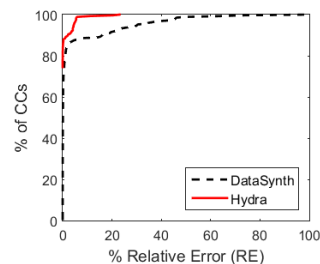


Figure 10: Quality of Volumetric Similarity (WL_c)

insert additional tuples to maintain referential integrity, the number is substantially smaller than those injected by DataSynth. This is because the integrity errors are *amplified* by the impact of the sampling errors. This effect is quantified in Figure 11, where the number of extra tuples inserted is plotted on a log-scale for representative TPC-DS tables. We see here that Hydra is often an *order-of-magnitude* smaller with regard to the addition of these extra tuples as compared to DataSynth. Also, recall that integrity errors in Hydra are independent of the data scale and therefore are minuscule at Big Data volumes. We also show this in [21].

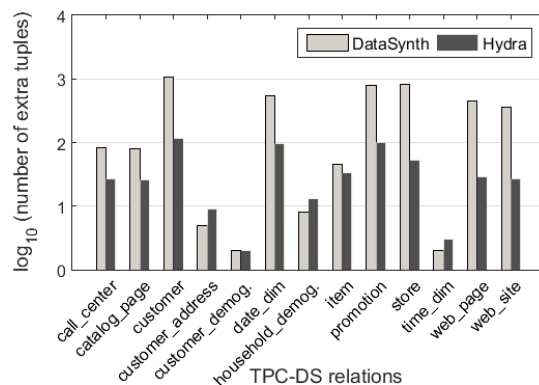


Figure 11: Extra tuples for Referential Integrity (WL_c)

As a final observation, it is interesting to note that DataSynth has to contend with both *negative* (volumes less than desired) and *positive* (volume greater than desired) relative errors, due to its sampling strategy – in fact, about one-third of the CCs suffered negative relative errors. In contrast, Hydra only generates positive errors due to the inclusion of extra tuples for satisfying referential integrity. From a practical standpoint, it is perhaps preferable to have positive errors since they induce greater stress on the data processing elements in the engine.

7.2 Scalability with Workload Complexity

We now turn our attention to evaluating the complexity of the underlying LP that is formulated by Hydra and DataSynth. Since LP complexity is essentially proportional to the number of variables in the problem, we compare this number for the two techniques. Further, since LP complexity is, to the first degree of approximation, independent of the database size, we present the comparison only for the 100 GB instance.⁴ The number of LP variables for a representative set of TPC-DS relations, including the major fact and dimension tables (`catalog_sales`, `store_sales`,

⁴Of course, the database engine’s choice of query plans may change to some extent with database size, leading to a slightly different set of CCs.

item) is captured, on a log-scale, in Figure 12 for the WL_c complex workload. We observe here that the LPs formulated using the region-partitioning strategy in Hydra generate *several orders of magnitude* fewer variables than the corresponding LPs derived from the grid-partitioning in DataSynth. As a case in point, consider the catalog_sales table – the number of variables created by DataSynth was almost 5.5 million, which is reduced to as low as 1620 by Hydra. Even more dramatic is the change for item table, where the number of variables is reduced from an enormous 10^{11} to around 3700.

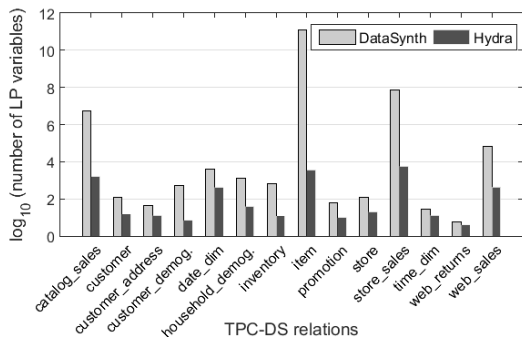


Figure 12: Number of variables in the LP (WL_c)

From an absolute perspective also, the large number of variables created by DataSynth is a critical problem since, as mentioned previously, the LP solver crashed in handling these cases. In marked contrast, the few thousands of LP variables generated by Hydra were easily solvable in less than a minute. Moreover, even when we switched to the simple workload, WL_s , the LP solution time for DataSynth was almost an hour, whereas Hydra completed in a few seconds as shown in Figure 13.

Complex Workload (WL_c)		Simple Workload (WL_s)	
DataSynth	Hydra	DataSynth	Hydra
crash	58 sec	50 min	13 sec

Figure 13: LP Processing Time

7.3 Scalability with Materialized Data Size

This experiment compares the data instantiation times, post LP solution, of DataSynth and Hydra on the WL_s workload. While Hydra, in principle, due to its summary-based approach, does *not* have to instantiate the data immediately, we assume in this experiment that the vendor requires complete materialization.

The experimental results are shown in Figure 14, where we also present, for comparative purposes, the performance with 10 GB and 1000 GB databases, apart from the default 100 GB database. We see here that there is a huge reduction in the materialization time of Hydra at all scales. Further, even in absolute terms, Hydra is able to output a 100 GB database in around 11 minutes, whereas DataSynth takes 42 hours to complete the same task.

The marked difference in the efficiency of the two techniques is attributed to the fact that DataSynth instantiates complete views through sampling, subsequently performs *several passes* on these instantiations to ensure referential integrity, and to derive relations from them. Hydra on the other hand, after LP-solving, constructs the database summary in just a few seconds, and then instantiates the materialized database directly from it.

Size (in GB)	DataSynth	Hydra
10	4 hours	2 min
100	42 hours	11 min
1000	> 1 week	1.6 hours

Figure 14: Data Materialization Time

7.4 Scalability to Big Data Volumes

In our next experiment, we validated the ability of Hydra, thanks to its summary-based technique, to scale to Big Data volumes. To demonstrate this feature, we modeled an exabyte-sized (10^{18} bytes) data scenario as follows: We used CODD, which is capable of modeling arbitrary metadata scenarios, to obtain the optimizer-chosen plans at the exabyte database scale for all the workload queries. To get AQPs for this database, we executed the obtained plans on the 100 GB instance and scaled the intermediate row counts with the appropriate scale factor. Hydra was able to formulate and solve the LPs (one per relation), and generate the database summary in less than **2 minutes**. Once the summary is generated, the database can begin to submit the workload queries since the data required for the execution can be produced on-the-fly by the Tuple Generator.

7.5 Dynamism in Data Generation

Our next experiment evaluates Hydra’s ability, due to the Tuple Generator and Database Summary architecture, to produce tuples *on-the-fly* instead of first materializing them, and then reading from the disk. To verify whether dynamic generation can indeed produce data at rates that are practical for supporting query execution, we compared the total time that Hydra’s tuple generator took to construct and supply tuples to the executor, while running simple aggregate queries, as compared to the standard sequential scan from the disk.

Rel. Name	Size (in GB)	Row count (in millions)	Scan time (secs)	
			Disk	Dynamic
store_returns	3	29	16	8
web_sales	10	72	43	25
inventory	19	399	107	74
catalog_sales	20	144	46	48
store_sales	34	288	168	87

Figure 15: Data Supply Times

The results of this experiment are shown in Figure 15 for the five biggest relations in the 100 GB database instance. We see here that the tuple generator is not only competitive with a materialized solution, but is in fact typically *faster*. Therefore, using dynamic generation can prove to be a good option since it can help to eliminate the large time and space overheads incurred in: (1) dumping generated data on the disk, and (2) loading the data on the engine under test.

7.6 Performance on JOB Benchmark

A legitimate concern with regard to the above encouraging results for Hydra is that they may be an artifact of the TPC-DS database, and perhaps might under-perform on other datasets. To address this concern, we consider in our final experiment, a schematically highly different database, namely the JOB benchmark [17], which is based on the IMDB real-world dataset. Here, we created a

workload of 260 queries, resulting in 523 CCs, whose cardinality distribution is again highly varied as seen in Figure 16.

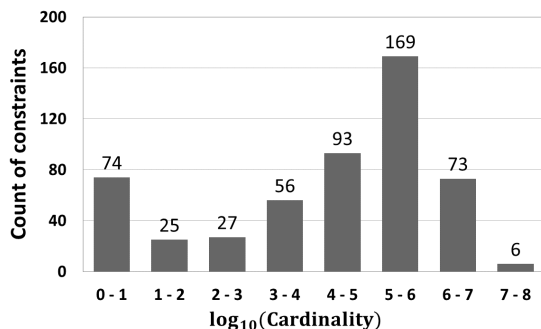


Figure 16: Cardinality distribution of CCs in JOB

We found that Hydra efficiently solved this workload as well, with the number of variables in each view being typically in the few thousands, and never exceeding a hundred thousand, as shown quantitatively in Figure 17. The overall database summary was quickly generated in around 20 seconds, and produced a database of high fidelity that satisfied all the constraints with no more than 2 percent relative error.

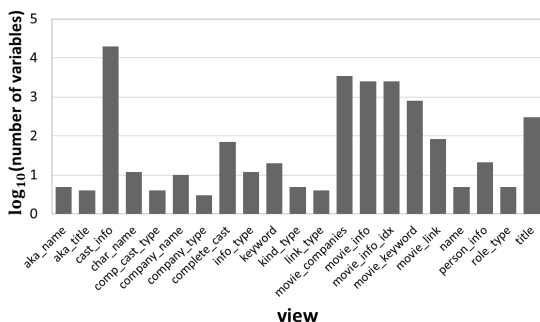


Figure 17: Number of Variables for JOB

8 RELATED WORK

Over the past few decades, a rich corpus of literature has developed on synthetic database construction. There are two broad streams of research on the topic, one dealing with the *ab initio* generation of new databases using standard mathematical distributions (e.g. [12, 15]), and the other with *regeneration* of an arbitrary existing database. In the latter category, there are two approaches, one of which uses only schematic and statistical information from the original database (e.g. [19, 22]). The other uses both the original database and the query workload to achieve statistical fidelity during evaluation (e.g [6, 11]) – our work on Hydra falls into this class. In this section, we briefly review recent literature on this spectrum of research categories.

Ab Initio Generation. Descriptive languages for the definitions of data dependencies and column distributions were proposed in [12, 16, 20]. For example, [12] proposed a special purpose language called Data Generation Language (DGL) that is used by the tool to generate synthetic data distributions by utilizing the

concept of iterators. It supports a broad range of dependencies between relations but the construction of dependent tables always requires access to the referenced table, creating a bottleneck on the data generation speed.

In contrast to the above, MUDD [23] and PSDG [16] generate all related data at the same time. However, this approach can also be rendered inefficient if the referenced tables are large in size. MUDD proposes algorithms to parallelize the data generation process, and to efficiently generate dense-unique-pseudo-random sequences and derive nonuniform distributions. Both MUDD and PSDG decouple data generation details from data description, facilitating customization of the tool to suit user needs.

In the distributed setting, a faster way of generating references is through recomputing since it eliminates the I/O costs incurred to satisfy referential constraints across relations that are present across different nodes. PDGF [20] was designed with this goal of achieving scalability and decoupling. In PDGF, the user specifies two XML configuration files, one for the data model and one for the formatting instructions. The generation strategy is based on the exploitation of determinism in pseudo-random number generators (PRNG), which enables regeneration of the same sequences, hence eliminating the scan overheads. PDGF supports the generation of data with cyclic dependencies as well, but incurs high computation costs for generating the associated keys. Finally, PDGF comes with a set of fixed generators for different datatypes and basic distribution functions.

A similar generator is Myriad [5], which implements an efficient parallel execution strategy leveraged by extensive use of PRNGs with random access support. With these PRNGs, Myriad distributes the generation process across the compute nodes and ensures that they can run independently from each other, without imposing any restrictions on the data modeling language.

Finally, a rule-based probabilistic approach, based on an extension of Datalog, has been recently proposed in [9], which is capable of generating data characterized by parametrized classical discrete distributions – however, it is not always feasible to assign such distributions to real-world data, especially over multivariate spaces.

Database-dependent Regeneration. DBSynth[19] is an extension to PDGF, which builds data models from an existing database by extracting schema information, and using sampling to construct histograms and dictionaries of text-valued data. Further, if the textual data contains multiple words, Markov chain generators are used to analyze the word combination frequencies and probabilities. Finally, after the model construction is complete, PDGF is invoked to generate the corresponding data.

Like DBSynth, RSGen [22] takes a metadata dump, including 1-D histograms, as the input, and generates database tables along with a loading script as the output. It uses a bucket based model at its core, which is able to generate trillions of records with minimum memory footage. However, the proposed technique works well only for queries with only a single range predicate. Further, due to the inaccurate statistical models in the query optimizer, the volumetric similarity is poor for queries involving predicates on correlated attributes.

UpSizeR [24] is a graph-based tool that uses attribute correlations extracted from an existing database to generate an equivalent synthetic database. A derivative work, Rex [13] produces an extrapolated database given an integer scaling factor and the original database, while maintaining referential constraints and the distributions between the consecutive linked tables. Dscaler [26]

addresses the problem of generating a non-uniformly⁵ scaled version of a database using fine-grained, per-tuple correlations for key attributes, but such information is typically hard to come by. Moreover, all these techniques only generate the *key* attributes, whereas the non-key values are sampled from the original database using these key values. Hence, the approach becomes impractical in Big Data and security-conscious environments. Finally, Dscaler fails to retain accuracy for some common query classes.

Query-dependent Regeneration. Apart from the above techniques, another line of work [6, 10, 11, 18] is based on workload dependence (as in the case of Hydra). Here the aim is to generate a database given a workload of queries such that volumetric similarity is achieved on these queries. In particular, RQP [10] gets a query and a result as input, and returns a possible database instance that could have produced the result for that query. The idea of using cardinalities from a query plan tree was first introduced in QAGen [11]. They start by constructing a *symbolic database*⁶, and then translate the input AQPs to constraints over the symbols in the database. Subsequently, a constraint satisfaction program (CSP) is invoked to identify values for symbols that satisfy all the constraints.

On the positive side, these generators are capable of handling complex operators as they use a general CSP, but the performance cost is huge since the number of CSP calls also increases with the database size. Further, it requires operating on a symbolic database of matching size to the original database, and processing of the entire database during the algorithm execution. This makes it impractical for Big Data environments. Finally, QAGen supports only one query plan in the input. This limitation was addressed in a follow-up tool called MyBenchmark [18], which creates a symbolic database on a per query basis and at the end tries to heuristically merge the various databases into a small number of databases. Clearly, generating a database on a per query basis has enormous time and space overheads, and further, a single database is not guaranteed in the output.

DataSynth [6] identified the declarative property of cardinality constraints and its ability to specify data characteristics. Given a large number of cardinality constraints as input, the paper proposed algorithms based on the LP solver and graphical models to instantiate tables that satisfy those constraints. However, it suffers from high LP complexity, data scale dependencies, and inaccuracies with regard to volumetric similarity, as we have discussed in this paper. Hydra materially extends the DataSynth approach by adding dynamism, scale and functionality.

9 CONCLUSIONS

The ability to synthetically regenerate data that accurately conforms to the volumetric behavior on queries at client sites is of crucial importance to database vendors, and will become even more so with the advent of Big Data applications. In this paper, we have proposed Hydra, a data regeneration tool that takes a substantial step forward towards achieving this goal. Specifically, by reworking the basic LP problem formulation into a region-based variable assignment, Hydra improves on the state-of-the-art DataSynth's performance by orders of magnitude with regard to problem complexity, data materialization time, and scalability to large volumes. Secondly, by using a deterministic alignment technique for database consistency, it provides far

better accuracy in meeting volumetric constraints as compared to the probabilistic approach employed in DataSynth. Finally, its summary-based framework organically supports the dynamic regeneration of streaming data sources, an essential pre-requisite for efficiently testing contemporary deployments.

In our future work, we plan to focus on covering a richer set of query operators, such as grouping functions, within the Hydra framework. Also, we would like to investigate how to leverage additional summary information (such as value-based correlations) that the client might be willing to provide for achieving stronger fidelity with the original database.

Acknowledgements. We thank the anonymous reviewers for their expert and constructive comments on the material presented here. We also thank Huawei Technologies India Pvt. Ltd. and the members of the Database Systems Lab at IISc for their valuable feedback and support in this work.

REFERENCES

- [1] TPC-DS. <http://www.tpc.org/tpcds/>.
- [2] TPC-H. <http://www.tpc.org/tpch/>.
- [3] USE PLAN SQL Server. [https://technet.microsoft.com/en-us/library/ms186954\(v=sql.105\).aspx](https://technet.microsoft.com/en-us/library/ms186954(v=sql.105).aspx).
- [4] PostgreSQL. <http://www.postgresql.org/docs/9.3/static/release.html>.
- [5] A. Alexandrov, K. Tzoumas and V. Markl. Myriad: Scalable and Expressive Data Generation. *PVLDB*, 5(12), 2012.
- [6] A. Arasu, R. Kaushik and J. Li. Data generation using declarative constraints. *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, 2011.
- [7] A. Arasu, R. Kaushik and J. Li. DataSynth: Generating synthetic data using declarative constraints. *PVLDB*, 4(12), 2011.
- [8] S. Ashoke and J. R. Haritsa. CODD: a dataless approach to big data testing. *PVLDB*, 8(12), 2015.
- [9] V. Barany, B. Cate, B. Kimelfeld, D. Olteanu and Z. Vagena. Declarative Probabilistic Programming with Datalog. *Proc. of the 19th Intl. Conf. on Database Theory*, 2016.
- [10] C. Binnig, D. Kossmann and E. Lo. Reverse Query Processing. *Proc. of the 23rd Intl. Conf. on Data Engineering*, 2007.
- [11] C. Binnig, D. Kossmann, E. Lo and M. Tamer Özsu. QAGen: generating query-aware test databases. *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, 2007.
- [12] N. Bruno and S. Chaudhuri. Flexible database generators. *Proc. of the 31st Intl. Conf. on Very Large Data Bases*, 2005.
- [13] T. S. Buda, T. Cerqueus, J. Murphy and M. Kristiansen. ReX: Extrapolating Relational Data in a Representative Way. *Proc. of the British Intl. Conf. on Databases*, 2015.
- [14] L. De Moura and N. Björner. Z3: An efficient SMT solver. *Proc. of the Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.
- [15] J. Gray, P. Sundaresan, S. Englert, K. Baclawski and P. J. Weinberger. Quickly Generating Billion-record Synthetic Databases. *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, 1994.
- [16] J. E. Hoag and C. W. Thompson. A parallel general-purpose synthetic data generator. *ACM SIGMOD Record*, 2007.
- [17] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper and T. Neumann. How Good Are Query Optimizers, Really? *PVLDB*, 9(3), 2015.
- [18] E. Lo, N. Cheng, W. W. K. Lin, W. Hon and B. Choi. MyBenchmark: generating databases for query workloads. *The VLDB Journal*, 23(6), 2014.
- [19] T. Rabl, M. Danisch, M. Frank, S. Schindler and H. Jacobsen. Just Can't Get Enough: Synthesizing Big Data. *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, 2015.
- [20] T. Rabl, M. Frank, H. M. Sergieh and H. Kosch. A Data Generator for Cloud-scale Benchmarking. *Proc. of the 2nd TPC Technology Conference on Performance Evaluation, Measurement and Characterization of Complex Systems*, 2010.
- [21] A. Sanghi, R. Sood, J. R. Haritsa and S. Tirthapura. Scalable and Dynamic Workload Dependent Data Regeneration. *Tech. Report TR-2017-01, DSL/CDS, IISc*, 2017. dsl.cds.iisc.ac.in/publications/report/TR/TR-2017-01.pdf.
- [22] E. Shen and L. Antova. Reversing statistics for scalable test databases generation. *Proc. of the 6th Intl. Workshop on Testing Database Systems*, 2013.
- [23] J. M. Stephens and M. Poess. MUDD: A Multi-dimensional Data Generator. *Proc. of the 4th Intl. Workshop on Software and Performance*, 2004.
- [24] Y. C. Tay, B. T. Dai, D. T. Wang, E. Y. Sun, Yong Lin and Yuting Lin. UpSizeR: Synthetically Scaling an Empirical Relational Database. *Inf. Syst.* 38(8), 2013.
- [25] R. S. Trivedi, I. Nilavalagan and J. R. Haritsa. Codd: Constructing dataless databases. *Proc. of the 5th Intl. Workshop on Testing Database Systems*, 2012.
- [26] J. W. Zhang and Y. C. Tay. Dscaler: Synthetically scaling a given relational database. *PVLDB*, 9(14), 2016.

⁵In non-uniform scaling, individual tables are scaled by different factors.

⁶A symbolic database is similar to a regular database, but its attribute values are symbols (variables), not constants.

TPStream: Low-Latency Temporal Pattern Matching on Event Streams

Michael Körber Nikolaus Glombiewski Bernhard Seeger

Database Systems Group, University of Marburg

{koerberm,glombien,seeger}@mathematik.uni-marburg.de

ABSTRACT

Complex Event Processing (CEP) has emerged as the state-of-the-art technology for continuously monitoring and analyzing streams of events in time-critical applications. The key feature in CEP is *sequential pattern matching* to detect a user-defined sequence of conditions on event streams. However, many CEP applications are not restricted to events only, but require native support for *situations* (aggregated event data lasting periods of time) and expressive temporal pattern matching among these situations. These important requirements regarding situations are not sufficiently addressed in the CEP literature so far.

In this paper we present *TPStream*, a novel event-processing operator for both deriving situations from event streams and detecting temporal patterns among situations. First, we provide a formal foundation of situations and *TPStream*. Then, we propose a low-latency algorithm for *TPStream* that delivers situations and temporal matches at the earliest possible point in time. Furthermore, we utilize a simple, yet effective cost model in order to adapt to changing workloads on the fly and with negligible cost for migrating operator states. The results of our experimental evaluation show that *TPStream* is capable of processing high-volume event streams with low latency and outperforms applicable CEP solutions from academia and industry.

1 INTRODUCTION

During the last decade, Complex Event Processing (CEP) has emerged to the technology of choice for analyzing massive streams of events in near real time. Typically, CEP systems detect composite (complex) events by combining, aggregating and filtering streams of simple or other composite events and report matches to registered event sinks. In turn, these sinks react to the detected event by triggering appropriate actions in a timely manner. CEP can be applied to a wide variety of application domains, including IT infrastructure monitoring, traffic monitoring, health care and financial applications.

Problem Statement: A noticeable fraction of currently available CEP systems is build upon point based temporal semantics. That is, each event is associated with a timestamp (e.g., the moment a measurement was made) and event streams are ordered accordingly. With only a *single* timestamp the expressible *temporal relations* between two events are limited to before/after/at the same time relationships. However, many real-world scenarios comprise the detection of complex *temporal patterns* among situations lasting for periods of time. Consider the following traffic-monitoring application:

Example: A traffic monitoring system is continuously receiving sensor data from connected cars (i.e., position, speed, acceleration). One of the systems goals is to notify drivers about potential

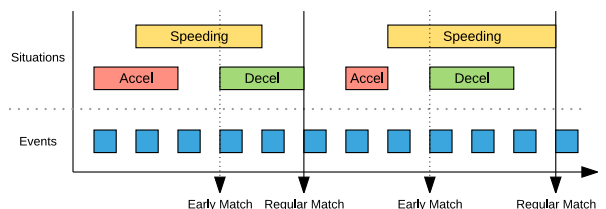


Figure 1: Detecting aggressive driving with situations

dangers around their locations, such as an aggressively driving car. Among others, the American Automobile Association has identified the following two actions being indicators for aggressive driving¹: “Operating the vehicle in an erratic, reckless, careless, or negligent manner or suddenly changing speeds” and “Driving too fast for conditions or in excess of posted speed limit”. From these definitions, a pattern to detect aggressive drivers could be stated as: “A sharp acceleration followed by hard braking, both accompanied by a period of speeding.”

Challenges: This example illustrates three key-features a solution for temporal pattern matching on instantaneous events should provide: (i) Situations are derived from point events on-the-fly, by identifying subsequences of the input stream for which the defined condition holds true (e.g., speed > 70 mph). Optional constraints can be applied to restrict the duration of situations. In addition to its validity (expressed as a time interval), a situation is enriched with meaningful summarizations of underlying events (e.g., the average speed of the speeding phase). (ii) The pattern language offers support for alternatives. That is, the order of the situations’ start and end points is not required to be fully fixed. Consider, for example, the two matches sketched in Figure 1: In the first match, the three defined situations *overlap*, while in the second match deceleration happens *during* the speeding situation. (iii) The pattern should be detected with the *lowest possible latency*. As depicted in Figure 1, both matches may be concluded at the beginning of the deceleration situation, since at this point in time speeding still holds true and the pattern allows any combination of their endpoints. Technically, this means the system should be able to conclude a successful match without exact knowledge about the validity of all situations.

State-of-the-Art: To the best of our knowledge, the only work on complex *temporal relations* in event stream pattern matching is the *ISEQ* operator [20]. However, *ISEQ* has several shortcomings concerning the desired features: First, the operator expects interval-events (i.e., situations) as input, leaving all aspects of (i) to an unspecified external entity. Being unaware of the origin of interval-events severely limits the operator in processing power (in terms of plan optimization) and most importantly renders a detection with the lowest possible latency (iii) impossible since there is no way to directly access an incomplete situation or indirectly manipulate the building of a situation through constraints.

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

¹<http://www.iii.org/fact-statistic/aggressive-driving>

Second, a *temporal pattern* is specified using a conjunction of endpoint relationships (i.e., an ordering on start (ts) and end (te) of intervals). This way, alternatives are expressed by omitting one or more endpoints. For example, the pattern $A.ts < B.ts < A.te \leq B.te \vee A.ts < B.ts < B.te < A.te$ on two situations A and B is expressed as $A.ts < B.ts < A.te$. Hence, disjunctions like $A.ts < B.ts < A.te < B.te \vee B.ts < A.ts < B.te < A.te$ are not expressible in a single query. Instead, they require multiple queries in an approach without any specified optimization component to detect shared processing opportunities. Finally, *ISEQ* relies on auxiliary index structures and punctuation mechanisms for efficient query execution, complicating the integration into existing systems.

Straw Man’s Approach: Besides *ISEQ*, we identified two approaches to solve the task of temporal pattern matching with point event streams. Thus, we can provide a point of comparison to CEP systems featuring pattern matching via regular expressions or equivalent techniques. The first approach works in two phases: In the first phase, a pattern matcher is deployed for each defined situation, computing its duration (start/end timestamp) and the desired aggregates. Technically, this means matching patterns of the form $!S S+ !S$ with S being derived from the input stream using the situation’s condition (e.g., $speed > 70 \text{ mph}$). This results in a dedicated stream per defined situation. Each of these streams is ordered according to the end timestamp, which allows to map the *temporal pattern* to a sequence of situations (reflecting the order of end timestamps, possibly containing alternatives). In the second phase, a dedicated pattern matching operator is used to find all matching sequences, whereby the proper ordering of start timestamps is checked via additional predicates. Even though this approach satisfies requirements (i) and (ii), it fails to produce early results (iii), because, just like in *ISEQ*, situations are fully derived before they are available for pattern matching.

The second approach uses a single pattern matching operator and expresses the *temporal pattern* as a single sequence of point events. To express temporal overlaps, the conditions of all involved situations must be connected via a logical AND. For example, *Acceleration overlaps Speeding* is expressed as $A B+ C$ with the following conditions: $A : accel > 8 \text{ m/s}^2$, $C : speed > 70 \text{ mph}$ and $B : A \wedge C$. Since patterns are expressed on the granularity of events, early results (iii) are achieved, by simply omitting the last portion of the pattern. At the same time summarizations of single situations and the validation of duration constraints (i) are left to a post-processing step, since situations are disassembled to express temporal overlaps.

Solution: We present *TPStream*, a holistic operator for complex temporal pattern matching on point event streams. Compared to the presented approaches, our contributions are:

- *TPStream* is the first CEP operator to closely couple derivation of situations with pattern matching, enabling match detection at the *earliest possible point in time*.
- We also improve upon existing work on temporal pattern matching (e.g. *ISEQ*) by allowing arbitrary alternatives and duration constraints in pattern definitions.
- We introduce an optimizer component for interval-based pattern matching which continuously adapts its execution strategy to deal with fluctuating data rates and changes in the data distribution of incoming streams.
- *TPStream* provides native query support for temporal pattern matching, making it easier to formulate and read *temporal*

patterns in comparison to most existing CEP solutions relying on a straw man’s approach.

- Unlike *ISEQ*, the operator and its low latency optimizations can easily be implemented in commonly available point-based systems, because time-intervals are used only internally and results are again point event streams.
- In experiments, we show our latency improvements and the performance limits of two existing CEP solutions from academia and industry when handling situations. We present that *TPStream* can outperform these systems by an order of magnitude and that alternatives, which have great impact on the performance of sequential pattern matching, influence *TPStream*’s matching performance only marginally.

The rest of the paper is organized as follows. Section 2 reviews related work, before we introduce *TPStream*’s query language in section 3. In section 4 we model all aspects of *TPStream* in an algebra. Efficient evaluation strategies, the algorithm for low-latency matching and our optimization techniques are presented in section 5. We evaluate the performance of *TPStream* in section 6 and conclude this paper in section 7.

2 RELATED WORK

So far, native ways to work with situations in systems capable of CEP have not been sufficiently addressed. Nevertheless, the concept can be related to working with time intervals, aggregating information and performing temporal joins, all of which have seen recent contributions. Since these are broad areas, we will loosely group the most relevant approaches under three headlines: Event Pattern Matching, Context/State in CEP and Spatio-Temporal Database Systems.

Event Pattern Matching: Systems capable of CEP (e.g. [3, 9]) are generally closely associated with a pattern matching operator. [27] features a discussion on the several different semantics of the operator and a recent survey [12] covers several implementations, that employ different techniques such as NFAs or Graphs, each featuring their own unique optimization techniques. Regardless of specific details, most approaches focus on data referring to points in time and thus lack *native* capabilities to query complex relations between time intervals as stated in [2] - a crucial aspect for dealing with long-lasting situations. Cayuga [8], ZStream [22] and Microsoft StreamInsight [1] are well-known approaches that associate time intervals with data, showcasing the interest in working with interval-based events. ZStream in particular shares similarities with our join-based, adaptive processing approach for pattern matching. However, each of the respective pattern languages is based around a strictly sequential relation (interval i ends before interval j begins) and/or explicitly order-independent relations (conjunction, disjunction). Not only does this limit their respective algorithmic support for complex *temporal relations*, but, just like point-based systems, formulating derivation queries naturally leads to the straw man’s approach mentioned above using Kleene Operators (or FOLDS in [8]). As we will show in our experiments, this approach results in significant performance deficiencies.

Context/State in CEP: There has been recent work on introducing *contexts* into a CEP environment. CAESAR [23] associates queries to long-lasting context windows, detects them from incoming events as soon as they start and suspends queries of inactive contexts. Similarly, contexts in [11] are used to group up event types to process them together. While contexts and

Relation (R)	Equivalent (R)	Visualization	Definition (δ_R)
A before B	B after A		$A.ts < A.te < B.ts < B.te$
A starts B	B started-by A		$A.ts = B.ts < A.te < B.te$
A meets B	B met-by A		$A.ts < A.te = B.ts < B.te$
A overlaps B	B overlapped-by A		$A.ts < B.ts < A.te < B.te$
A during B	B contains A		$B.ts < A.ts < A.te < B.te$
A finishes B	B finished-by A		$A.ts < B.ts < A.te = B.te$
A equals B			$A.ts = B.ts < A.te = B.te$

Table 1: Allen’s Interval Algebra

situations are related concepts, the key difference is that contexts are purposefully decoupled from events. Therefore, it is not possible to query the relation of different contexts to each other. In contrast, *TPStream* focuses on efficient, adaptive and low-latency implementations of those *temporal relations*. Likewise, work on states [18] and on aggregating windows [14, 17] focuses on derivation, but lacks interval relations [2] or pattern matching.

Spatio-Temporal Database Systems: The spatial databases community studied the problem of *spatio-temporal pattern queries* (STPQ) in trajectory databases [10]. In general, these approaches cannot be directly applied to an event processing environment, because they are built on top of a *persistent* trajectory database model, where movement histories are already stored and indexed in the database. However, the design of [25] in particular served as a foundation for our proposed *TPStream* operator as *TPStream* adapts similar concepts of temporal predicates and constraints. Furthermore, our evaluation method is related to temporal joins (see [13] for an excellent survey), but as most of the work is not based in stream processing, unique and important issues such as continuously arriving data, continuous query optimization and early result detection are overlooked. In comparison to join algorithms on streams [6, 15] as well as adaptive approaches [5], *TPStream* combines both the derivation of situations and the detection of patterns. Thus, the operator can offer new techniques for early result detection unique to CEP-style pattern matching.

3 QUERY LANGUAGE

In order to express *temporal relations* between situations, we adopt Allen’s Interval Algebra [2] depicted in Table 1 for two generic intervals A and B. Each interval has a starting point (ts) and an ending point (te), resulting in a total of four points. te is the first point in time when the interval is not valid, i.e. the interval is *half-open*. The *relation* (R) between A and B is represented through the relation between these four points as given by the *definition* (δ_R). As an example depicted in Table 1 A *before* B means the interval A ends before the interval B begins. Similarly, A *during* B means A happens during B, because A.ts and A.te are between both points of B. We introduce the *TPStream* query language by formulating and explaining the query to detect aggressively driving cars from the introductory example in Listing 1.

The operator works on streams containing data referring to points in time. In the case of aggressive drivers, we work on a singular stream CS, providing sensor data from cars, which is specified as an input (FROM). This stream is partitioned by the car_id to evaluate each driver individually (PARTITION BY). The important aspect of deriving situations from the stream is handled in the DEFINE clause: The acceleration situation is represented with the symbol A, the condition $CS.accel > 8 \text{ m/s}^2$ and the

```

FROM CarSensors CS PARTITION BY CS.car_id
DEFINE A AS CS.accel > 8m/s2 at least 5s,
       B AS CS.speed > 70 mph between 4s AND 30s,
       C AS CS.accel < -9m/s2 at least 3s
PATTERN A meets B; A overlaps B; A starts B; A during B
        AND C during B; B finishes C; B overlaps C; B meets C
        AND A before C
WITHIN 5 MINUTES
RETURN first(B.car_id) AS id,
        avg(B.speed) AS avg_speed;

```

Listing 1: Aggressive drivers query

(optional) duration constraint AT LEAST 5s, while speeding and deceleration are defined by B and C respectively. The derived situations are analyzed with a PATTERN. For aggressive drivers, an acceleration (A) may meet, overlap, start or occur during a phase of speeding (B). These are alternatives in the pattern definition, separated with semicolons in the query language. The same applies for deceleration (C) and speeding (B). The pattern is fulfilled if at least one of *each* alternatives is true. We apply a window condition on the evaluation period (WITHIN), specifying that the pattern should only be searched within situations derived in the past 5 minutes. Finally, in case of a match, we RETURN aggregated results from each situation, in this case the car_id and the average speed.

3.1 Expressiveness

Most common CEP systems define patterns based on symbols connected via regular expressions. Specific extensions, like aggregations, put the expressiveness of those languages between regular and context-free grammars [27]. However, only *ISEQ* provides a *native* way to process patterns based on *temporal relations*. This deficit is also reflected in the respective languages.

By design, *TPStream* can express all *temporal relations* (and unlike *ISEQ* alternatives among them) in a *single* query. In contrast, a single pattern matching query in CEP systems is designed to detect a sequence, i.e., a *before* relation. Nevertheless, as shown by both straw man’s approaches in our introduction, in a system supporting Kleene-closure it is possible to express other *temporal relations* through either multiple queries (decoupling derivation and detection) or a single query (without aggregation capabilities and the validation of duration constraints). Thus, our language does not express more than the *full* language of other systems.

Instead, we focus on enabling the user to express complex *temporal patterns* in a single, readable and maintainable query via the widely-known interval algebra (Table 1). For this purpose, we made two notable design choices that differ from some sequence-based approaches. First, some languages [27] allow to skip events while matching. In contrast, we derive the longest possible contiguous sequence of events, because this aligns well with the idea of long lasting situations and avoids ambiguity whether a situations is still ongoing during *other* events. Second, some languages [9] allow symbols to access aggregates of other symbols. Due to ambiguity in the expected results when dealing with situations, we do not allow this. For example, consider modifying the definition for symbol B in Listing 1 to $B \text{ AS } CS.speed > \max(A.speed)$. Then, for A overlaps B it is unclear whether $\max(A.speed)$ is accessed when A finishes, when B starts or is continuously monitored for each B. For a precise presentation of our approach, we chose those two concessions and will work on mitigating them in the future.

We would also like to sketch that, apart from those concessions, it is possible to express a purely sequence-based pattern with *TPStream*: A sequence can be expressed with a before relation and the implicit ongoing nature of situations can be eliminated with a duration constraint. Nevertheless, the basis for our implementation [19] features a standard sequence-based pattern matching operator that is optimized and thus preferable for this purpose. Similarly, our implementation can be easily integrated into other systems, because *TPStream* consumes and produces point-based event streams. In conclusion, this means that we do not change the expressiveness of other approaches, but by extending a query language with Allen’s Interval Algebra, our benefits can be almost universally adopted.

4 ALGEBRA

The goal in designing *TPStream* is to develop an operator capable of continuously deriving situations from a stream of events and relate those situations to each other. To achieve this, we need to be able to express both the derivation and relation. For this purpose, we will formally model those aspects (streams, data, deriving situations and temporal pattern matching) in an algebra.

4.1 Stream Model

Definition 1 (Data Stream). A data stream D is a potentially unbounded sequence of data items $\langle d_1, d_2, \dots \rangle$ totally ordered by a relation $<_D$. $d_i \in D$ refers to the i -th data item in the stream according to that order and all data items are from the same domain \mathcal{D} . \diamond refers to an empty data stream.

In order to refer to multiple data streams, we will utilize the notation D^1, D^2, D^3, \dots with $D^i = \langle d_1^i, d_2^i, \dots \rangle$, i.e. a superscript labels separate streams, while a subscript refers to the order within a stream. For the sake of simplicity and legibility, we will generally assume that each item in a data stream is unique and refer to previous work on the matter of handling potentially equal elements [8]. \diamond is mainly used to specify the case of *no output* in upcoming definitions.

Definition 2 (Continuous Subsequence). Based on a data stream D , $D_{[i,j]} = \langle d_i, \dots, d_j \rangle$ with $i < j$ refers to a continuous subsequence containing every data item as it pertains to $<_D$.

Definition 3 (Union). The union \uplus of two data streams D^1 and D^2 both totally ordered with $<_D$ results in a data stream D' with the same order $<_D$:

$$\uplus(D^1, D^2) := D' = \langle d'_1, d'_2, \dots \rangle$$

such that D' contains each element from both D^1 and D^2 . Analogous to set theory, the union of n data streams D^1, \dots, D^n is abbreviated with the notation $\biguplus_{i=1}^n D^i$.

4.2 Data Model

Our operator involves two kinds of data which we need to define: *events* and *situations*. In general, events refer to a notification that something happened instantaneously at exactly one point in time while situations span multiple points in time and contain aggregated information for that time period.

Definition 4 (Event). An event e is a pair (p, t) consisting of a payload p and an event timestamp t . p is from some domain \mathcal{D} and t is from a discrete and totally ordered time domain \mathcal{T} . The validity of p is the instant t .

Definition 5 (Situation). A situation s is a triple (p, ts, te) consisting of a payload p and two timestamps: ts (start timestamp) and te (end timestamp). p is from some domain \mathcal{D} . ts and te are from a discrete and totally ordered time domain \mathcal{T} with $ts < te$. The half-open time interval $[ts, te)$ specifies the validity of p .

Event streams are ordered by the event timestamp and will be represented with E . **Situation streams** are ordered by the end timestamp of situations and will be represented with S . We focus our efforts on presenting algorithms for streams with data arriving in-order and leave the adjustment to out-of-order data by adapting previous research on out-of-order pattern matching [7, 21] for future work.

4.3 Derivation

Situations are derived from event streams through aggregation and predicate evaluation. We will first formally define aggregation on continuous event subsequences before deliberating on predicates and how to derive situation streams.

Definition 6 (Aggregated Event Subsequence). An aggregate γ_{agg} is applied to an event stream subsequence $E_{[i,j]}$ by applying the aggregate agg to the events in the subsequence:

$$\gamma_{agg}(E_{[i,j]}) := (agg(e_i, \dots, e_j), e_i.ts, e_{j+1}.ts)$$

When obvious from context, we abbreviate γ_{agg} with γ .

The result in Definition 6 technically already is a situation. However, for the derivation process as a whole, we want to discover situations for which a set of circumstances hold true. In order to provide an unambiguous process to identify these situations we are looking for the longest possible sequences for which these circumstances apply.

Definition 7 (Derived Situation). Situations are derived with a function $derive_{\phi, \tau, \gamma}$ which aggregates information of a continuous event subsequence $E_{[i,j]}$ by applying γ iff the events in $E_{[i,j]}$ are the longest possible sequence of events to fulfill a given predicate ϕ and the covered timespan is within the given duration constraint $\tau := [d_{min}, d_{max}]$:

$$derive_{\phi, \gamma, \tau}(E_{[i,j]}) = \begin{cases} \forall l \in [i, j] : \phi(e_l) \wedge \\ \gamma(E_{[i,j]}) & \text{if } !\phi(e_{i-1}) \wedge !\phi(e_{j+1}) \wedge \\ & (e_{j+1}.ts - e_i.ts) \in \tau \\ \diamond & \text{otherwise} \end{cases}$$

Example. Assume the query in Listing 1 derives a speeding situation for a car with the time interval $[2, 10)$. This means $CS.speed \leq 70$ mph at $t = 1$ and $t = 10$ and in between those timestamps $CS.speed > 70$ mph. From an algebraic standpoint, assuming knowledge about the whole event stream, this aligns well with a natural interpretation: There are not multiple situations (e.g. $[2, 3), [2, 4), \dots$) but rather one continuous speeding phase which fulfills the duration constraint ($d_{min} = 4s$ and $d_{max} = 30s$). For that reason and because it results in unique situations, we choose to derive the longest possible subsequence in Definition 7.

Definition 8 (Derived Situation Stream). The $deriveStream_{\phi, \gamma, \tau}$ function derives a stream of situations from a given event stream E by applying the function $derive_{\phi, \gamma, \tau}$ to all possible subsequences and unifying the results:

$$deriveStream_{\phi, \gamma, \tau}(E) = \biguplus_{j=1}^j \biguplus_{i=1}^j derive_{\phi, \gamma, \tau}(E_{[i,j]})$$

Note that, due to assumption that each event in an event stream has a unique timestamp and the fact that $derive_{\phi, \gamma, \tau}$ derives the longest situations possible, it is easy to show that $deriveStream_{\phi, \gamma, \tau}$ produces a stream of situations with disjoint time intervals. This implies, that the order of situations using start timestamps is the same as the order using end timestamps, resulting in a beneficial pattern for query processing [16]. Due to space limitations, we omit a formal proof here.

4.4 Pattern Matching

TPStream matches multiple situation streams to a *temporal pattern* and produces a result event stream according to the given definitions. A *temporal pattern* is composed of *temporal constraints* between situation streams, which in turn comprise multiple *temporal relations* between exactly two streams. In this section, we present formal definitions of these terms, the output of a successful match and ultimately the *TPStream* operator.

Definition 9 (Temporal Relation). Given two situation streams S^A, S^B , a *temporal relation* $R^{A,B}$, defines a valid relationship between two situations $s^A \in S^A$ and $s^B \in S^B$ according to Allen's Interval Algebra (cf. Table 1). s^A and s^B fulfill $R^{A,B}$, iff they satisfy the corresponding algebraic definition (δ_R).

Definition 10 (Temporal Constraint). A *temporal constraint* $C^{A,B}$ between two situation streams S^A, S^B is a set of *temporal relations* $\{R_1^{A,B}, \dots, R_m^{A,B}\}$. Two situations $s^A \in S^A$ and $s^B \in S^B$ fulfill $C^{A,B}$, iff they at least fulfill one of the *temporal relations*.

In other words, *temporal constraints* allow to specify multiple valid relations between two situation streams, providing the desired flexibility in expressing alternatives.

Definition 11 (Temporal Pattern). For any number of situation streams (S^1, \dots, S^m) , a *temporal pattern* (\mathcal{P}) is a set of *temporal constraints* $\{C^{i,j} | 1 \leq i < j \leq m\}$. A *temporal pattern* is matched by a *temporal configuration* $\bar{s} = (s^1 \in S^1, \dots, s^m \in S^m)$, iff \bar{s} satisfies every *temporal constraint*:

$$match_{\mathcal{P}}(\bar{s}) : \Leftrightarrow \forall C^{i,j} \in \mathcal{P} : \exists R^{i,j} \in C^{i,j} : \delta_{R^{i,j}}(s^i, s^j)$$

Example. Consider the example query of Listing 1 and let s^A be an acceleration situation as defined by A and s^B, s^C be a speeding (B) and deceleration (C) situation respectively. The PATTERN describes how pairs of situations in $\bar{s} = (s^A, s^B, s^C)$ can relate to each other via *temporal constraints*: For s^A and s^B the *temporal relation* can be either A meets B, A overlaps B, A starts B or A during B. It does not matter if acceleration overlaps speeding or if speeding contains acceleration. Both cases may lead to the result of detecting aggressive drivers. The *temporal pattern* on the other hand is a conjunction of *temporal constraints*: In order to match the pattern, each *temporal constraint* must be fulfilled.

Definition 12 (Pattern Matching Output). A temporal pattern matching operator $PM_{w, \hat{\gamma}}$ matches a *temporal configuration* $\bar{s} = (s^1, s^2, \dots, s^m)$ to a *temporal pattern* \mathcal{P} . It aggregates the information of \bar{s} with some suitable aggregate $\hat{\gamma}$ and checks the *window* condition (cf. WITHIN clause):

$$window(\bar{s}, w) = w \leq \max_{s \in \bar{s}}(s.te) - \min_{s \in \bar{s}}(s.ts)$$

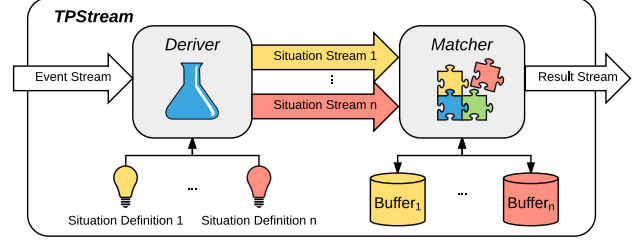


Figure 2: *TPStream* Architecture

The operator produces an output, if the *temporal configuration* matches the pattern during the specified window, i.e.:

$$PM_{w, \hat{\gamma}}(\bar{s}, \mathcal{P}) := \begin{cases} (\hat{\gamma}(\bar{s}), \max_{s \in \bar{s}}(s.te)) & \text{if } match_{\mathcal{P}}(\bar{s}) \wedge window(\bar{s}, w) \\ \diamond & \text{otherwise} \end{cases}$$

Similarly to how we extended derived situations to derived situation streams (Definition 7 to 8), we can extend Definition 12 to situation streams:

Definition 13 (*TPStream*). $TPStream_{w, \hat{\gamma}}$ matches multiple situation streams S^1, \dots, S^m to a *temporal pattern* \mathcal{P} by applying the corresponding pattern matching operator $PM_{w, \hat{\gamma}}$ to the cross product of the situation streams and unifying the results:

$$TPStream_{w, \hat{\gamma}}(S^1, \dots, S^m, \mathcal{P}) := \bigcup_{\bar{s} \in \times_{i=1}^m S^i} PM_{w, \hat{\gamma}}(\bar{s}, \mathcal{P})$$

Note that $TPStream_{w, \hat{\gamma}}$ results in an event stream and can thus easily be integrated into common CEP processing pipelines.

5 ALGORITHMS & IMPLEMENTATION

In this section, we present our algorithms and implementation details for detecting *temporal patterns* among streams of point events. Following the definitions from the previous section, the general architecture consists of two main components, as depicted in Figure 2. The *deriver-component* consumes events from the input stream and derives the defined situation streams. Then, those streams are passed to the *matcher-component*, which performs the actual pattern matching. In the following two subsections we will explain both components in detail. For the sake of simplicity we defer low latency detection to section 5.3 and initially wait for the end timestamp of derived situations before invoking the *matcher*. The last part of this section describes how *TPStream* computes efficient execution plans and dynamically adapts to changing workloads.

5.1 Deriving Situations

Definition 8 introduced derived situation streams, using knowledge about the whole input-stream. To compute situation streams incrementally as new events arrive the *deriver-component* manages a buffer for ongoing situations (B) and the situation stream definitions (D). Algorithm 1 shows how they are used to derive situations on-the-fly. For each defined situation, 3 cases are checked: If there is no started situation on the buffer, but the predicate holds true, a new situation is started. Therefore, we compute initial values for all defined aggregates (e.g., p.speed for an $\max(\text{speed})$ aggregate). Those values are bundled with the event's timestamp and stored on the buffer (Lines 4,5). The

Algorithm 1: DeriveSituations

Input: (p, t) : event
Data: $B := [(p', ts)_i]$: active situation buffer,
 $D := [(\phi, \gamma, \tau)_i]$: situation definitions

```
1  $R \leftarrow \emptyset$ ;  
2 foreach  $i \in |D|$  do  
3    $(\phi, \gamma, \tau) \leftarrow D[i]$ ;  
4   if  $B[i] = \emptyset \wedge \phi(p)$  then  
5      $B[i] \leftarrow (initAgg(p, \gamma), t)$ ;  
6   else if  $\phi(p)$  then  
7      $updateAgg(p, B[i], \gamma)$ ;  
8   else if  $B[i] \neq \emptyset$  then  
9     if  $(t - B[i].ts) \in \tau$  then  
10       $R \leftarrow R \cup \{(B[i].p', B[i].ts, t)\}$ ;  
11       $B[i] \leftarrow \emptyset$ ;  
12 if  $R \neq \emptyset$  then  
13    $updateMatcher(R, t)$ ;
```

Algorithm 2: UpdateMatcher

Input: S : set of finished situations, t : the current time

```
1  $purgeBuffers(t)$ ;  
2 foreach  $s \in S$  do  
3    $addToBuffer(s)$ ;  
4    $performMatch(\{s\}, 0)$ ;
```

temporal validity of a started situation is prolonged, if the current event fulfills the predicate. In this case, the buffered aggregates are updated using the event's payload (p) (Lines 6,7). Finally, a situation is finished on the first event not satisfying the defined predicate. In this case, the situation's end timestamp is fixed to the current time, it is added to the result-set R (provided it satisfies the duration constraint τ) and the corresponding buffer slot is cleared (Lines 8-11). After updating the state of each situation stream, the result-set is passed to the *matcher-component* (Lines 12,13).

5.2 Matching the Pattern

The *matcher* implements an incremental version of Definition 13 ($TPStream_{w, \hat{\gamma}}$). In other words, it detects matches on-the-fly as new situations are handed over from the *deriver-component*. The general idea is to employ a buffer for each situation stream and perform the pattern detection via a multi-way join between those buffers, using the *temporal constraints* as join-conditions. Recap that all situations within a stream are disjoint and thus imply the same order on both the start and end timestamps (Definition 8). We will use this fact to ensure efficient execution of the matcher component.

Each time the *deriver* distills new situations, Algorithm 2 is invoked: At first, expired situations are purged from the buffers (Line 1). That is, removing all situations s with $s.ts < t - window$. Because of the mentioned ordering, this effectively means, finding the first situation s' with $s'.ts \geq t - window$ and discarding all previous events. The buffers are implemented via array-backed ring buffers, which efficiently support these operations.

After purging outdated situations, each new situation is first added to its corresponding buffer, before the actual matching

Algorithm 3: PerformMatch

Input: ws : working-set, sc : current step count
Data: order: evaluation order

```
1 if  $sc = order.getNumSteps()$  then  
2    $publishResult(ws)$ ;  
3   return;  
4  $step \leftarrow order.getStep(sc)$ ;  
5 if  $step.isSet(ws) \wedge step.checkConstraints(ws)$  then  
6    $performMatch(ws, sc + 1)$ ;  
7 else if  $!step.isSet(ws)$  then  
8   foreach  $(p, ts, te) \in findMatches(step, ws)$  do  
9      $ws \leftarrow ws \cup \{(p, ts, te)\}$ ;  
10     $performMatch(ws, sc + 1)$ ;  
11     $ws \leftarrow ws \setminus \{(p, ts, te)\}$ ;
```

algorithm (Algorithm 3) is invoked (Lines 2-4). We force the new situation to be part of any successful match, by passing it as a parameter. This ensures the desired incremental creation of results, because we pass a new, not yet considered situation on every invocation.

The matching algorithm relies on a so called *evaluation order*, which we describe briefly upfront. An *evaluation order* determines the order in which situation buffers are joined and provides the required information for each processing step (a reference to the situation buffer and the set of *temporal constraints* to be fulfilled). Using this information and a partial *temporal configuration* (*working-set*), Algorithm 3 matches the *temporal pattern* as follows: In each step, the corresponding situation buffer is searched for situations satisfying all applicable *temporal constraints* (Line 8). Applicable means, that the counterpart of the constraint is already present in the *working-set*. Then, all returned situations are successively added to the *working-set* and for each of the new partial *temporal configurations*, the algorithm proceeds to the next step (Lines 9-11). Lines 5 and 6 intercept the evaluation, if the *working-set* already contains a situation for the current step, which accounts for situations passed as a parameter from Algorithm 2. In this case, the corresponding buffer is ignored and the step's *temporal constraints* are checked directly. Finally, a match is detected if the *working-set* contains a situation from every buffer (Lines 1-3). The *publishResult* function consumes this *working-set*, assembles the result and pushes it into the output stream.

Obviously, the evaluation performance of Algorithm 3 mainly depends on the efficiency of the *findMatches* function. A naïve approach would be, to scan the entire buffer and check the *temporal constraints* for each situation separately. With R_i denoting the i -th intermediate result, B_i the buffer traversed in step i and $|R_i| = |B_i|$, the costs (C) of *performMatch* following this approach can be estimated with:

$$C = |R_n| + \sum_{i=1}^{n-1} |R_i| \cdot |B_{i+1}| \quad (1)$$

To speed up the computation, we again use the order of situation streams: Because the order is reflected on the buffers, we are able to find all matching situations using binary searches.

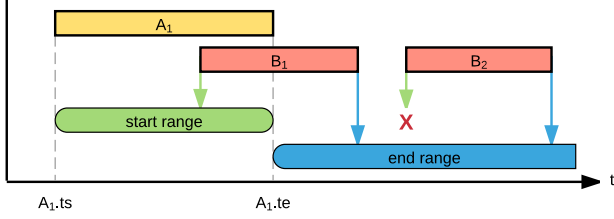


Figure 3: Temporal Matching via Range Queries

We first discuss how this is done for a single *temporal relation* before extending it to (multiple) *temporal constraints*. Recall that a *temporal relation* explicitly defines a relationship between all four endpoints of two situations. For instance, this is $A.ts < B.ts < A.te < B.te$ for A overlaps B. Now, given an instance of situation A, we can obtain matching instances of B by (i) issuing two range-queries on the buffer of B, using the timestamps of A as boundaries and (ii) intersecting the results of those queries. For the example relation, these queries are:

- (1) $A.ts < ts < A.te$ for the start-timestamp and
- (2) $A.te < te < \infty$ for the end timestamp.

It is easy to see, that each situation falling into both ranges fulfills the given *temporal relation*. Figure 3 illustrates this using 3 situations: Situation A_1 in combination with the *temporal relation* is used to build the two search ranges. After intersecting the results ($\{B_1\}$ for the start range and $\{B_1, B_2\}$ for the end range), we receive our final result B_1 . Note that for *temporal relations* allowing more than one result (e.g., A before B), this strategy additionally eliminates the need for checking each combination individually.

Typically, a *temporal constraint* contains more than one *temporal relation*, stating each of them as a valid relationship between two situations. This can be easily integrated by executing the search separately for each of the defined relations and subsequently building the union of the obtained results. The conjunction of multiple *temporal constraints* can be implemented as an intersection of the results from the respective individual queries. Because the buffers are backed by a contiguous array, we can represent the search results as index-ranges and thus efficiently compute the required unifications and intersections. This approach reduces the estimated costs of *performMatch* to:

$$C = \sum_{i=2}^n \left(|R_{i-1}| \cdot |R_i| + C_{findMatches}(|B_i|) \right) \quad (2)$$

with $C_{findMatches}(|B_i|)$ being bounded by $|\mathcal{P}| \cdot 13 \cdot 4 \cdot \log_2(|B_i|)$. The constant factors 13 and 4 arise from the possible *temporal relations* per constraint and the binary searches to execute for each of them, respectively.

5.3 Low-Latency Matching

In this section, we will determine the earliest points in time (t_d) to detect a *temporal relation* ($t_d(R)$), *temporal constraint* ($t_d(C)$) and *temporal pattern* ($t_d(\mathcal{P})$). Then, we adjust our algorithms from the previous section to deliver matches as early as possible.

5.3.1 Analysis. Two situations A, B can only be related once we know they exist, making $\max(A.ts, B.ts) \leq t_d(R)$ a trivial lower bound for *all* relations. For exact $t_d(R)$ consider a relation's definition δ_R depicted in Table 2. Let $t_1 \leq t_2 \leq t_3 \leq t_4$ be the timestamps in the order they appear in δ_R . It is easy to see

Relation (R)	Definition (δ_R)	$t_d(R)$	Prefix-Group (G)	$t_d(G)$
A before B	$A.ts < A.te < B.ts < B.te$	B.ts		
A meets B	$A.ts < A.te = B.ts < B.te$	B.ts	$A.ts < A.te \leq B.ts$	B.ts
A starts B	$A.ts = B.ts < A.te < B.te$	A.te		
A equal B	$A.ts = B.ts < A.te = B.te$	$A.te = B.te$	$A.ts = B.ts$	B.ts
A started-by B	$A.ts = B.ts < B.te < A.te$	B.te		
A overlaps B	$A.ts < B.ts < A.te < B.te$	A.te		
A finishes B	$A.ts < B.ts < A.te = B.te$	$A.te = B.te$	$A.ts < B.ts$	B.ts
A contains B	$A.ts < B.ts < B.te < A.te$	B.te		

Table 2: Low-Latency Analysis

that the ordering of t_4 can implicitly be derived at t_3 , because $t_3 \leq t_4$ and there are no timestamps beyond that. Furthermore, at t_1 and t_2 there are other relations sharing the same definitions up to that point, i.e., it is not possible to distinguish them from each other. To show this, we have grouped relations starting with $A.ts$ as *prefix groups* in Table 2 (B.ts groups are analogous). For those reasons we can conclude $t_d(R) = t_3$.

A *temporal constraint* $C = (R_1, \dots, R_n)$ for A, B matches if at least one relation matches. Therefore, the earliest detection time is $t_d(C) = \{t_d(R_1), \dots, t_d(R_n)\}$. Note that $t_d(C)$ is a set and the actual detection time of two situations depends on the fulfilled relation. Further, if C contains *all* relations of a *prefix group* (cf. Table 2), the detection time of these relations is shifted to the trivial lower bound ($t_d(G)$).

Finally, for a pattern $\mathcal{P} = (C_1, \dots, C_m)$, each constraint must be matched. However, a single *temporal configuration* matching \mathcal{P} fulfills exactly one *temporal relation* (\bar{R}) from each constraint, making $t_d(\mathcal{P}) = \max(t_d(\bar{R} \in C_1), \dots, t_d(\bar{R} \in C_m))$. In general, $t_d(\mathcal{P})$ is among the constraint detection points: $t_d(\mathcal{P}) \subseteq \bigcup_{i=1 \dots m} t_d(C_i)$.

5.3.2 Implementation. For the ease of presentation, we ignore the optional duration constraints on situations as well as *prefix groups* during the development and discuss the required changes at the end of this section. We gained two implementation-relevant insights from the low-latency analysis. First, new matches can only be detected if a new situation starts or a situation ends. Second, only a subset of the defined situations can possibly produce a match at $t_d(\mathcal{P})$. Thus, the matching process can be delayed until a situation with at least one endpoint in $t_d(\mathcal{P})$ occurs without affecting the latency. We call those situations *trigger situations* since only they should *trigger* a *performMatch* call. These insights affect our algorithms in the following ways. Situations must be available for matching from their start on, which can easily be achieved by adjusting the *deriver*. Additionally, we need to determine for each situation stream if the derived situations are *trigger situations*. For *trigger situations* we need to identify the point in time to execute *performMatch* (at its start, end or both).

However, the following cases must be considered during the matching process. If a situation requires matching on both endpoints, care must be taken not to produce duplicate results. Further, started situations must not be visible to the matcher in all cases: If two situations are related via *finishes* or *equals*, they could be mistakenly matched, because their temporary end timestamps (i.e., the current time) are equal. On the other hand, if two situations are not explicitly related via a *temporal constraint* they may participate in a successful match, even if both end timestamps are unknown. To illustrate this, consider the following pattern on four situations (A, B, C, D): A before B AND A before C AND A before D AND (D during C OR C

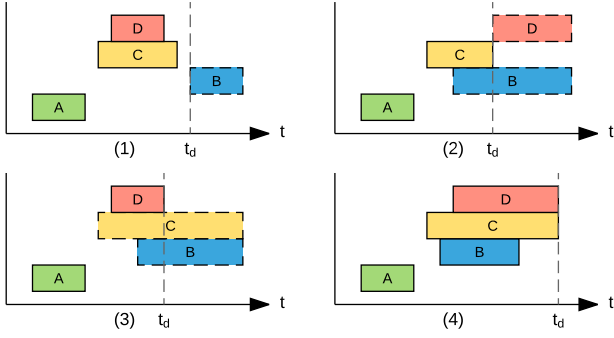


Figure 4: Earliest detection time (t_d) of different temporal configurations for the same pattern

finishes D OR C meets D). It defines situation A as starting point of every match. B is not explicitly related to C and D and required to happen after A. Consequently, B is a *trigger* situation and B.ts is in $t_d(\mathcal{P})$. For D both D.ts (via meets) and D.te (via during, finishes) are in $t_d(\mathcal{P})$. Figure 4 shows four representative *temporal configurations* for this pattern, highlighting the earliest point of detection (t_d). Configuration 1 showcases B as a *trigger* situation with $t_d = B.ts$. For the second configuration, D is the *trigger* with $t_d = D.ts$. This case also shows that two started relations may participate in a match if their end is unknown (B, D). The remaining configurations highlight D as a *trigger*, but with $t_d = D.te$.

Instead of handling these cases explicitly, our low latency algorithm avoids them by ensuring a unique combination of situations in the *working-set*, before passing it to the matching algorithm. In particular, this means started situations are managed in a separate buffer, inaccessible for the matching algorithm, and all valid combinations among them (i.e., all combinations of started situations, not explicitly related to the current one) are built upfront inside the *working-set*. Furthermore, to avoid duplicate results, the following fact is exploited: *temporal relations* enforcing matching on a situation's start require its counterpart to be finished in the past. On the other hand, *temporal relations* triggering matching on a situation's end require its counterpart to be either started (and not yet finished) or finished at the same time (cf. Table 2). Consequently, manually adding the started counterpart to the *working-set*, before executing the matching algorithm on a situation's end ensures uniqueness of the produced results.

The details are presented in Algorithm 4. After purging outdated situations from the buffers (Line 1), each started situation (s) is added to the additional buffer and if $s.ts \in t_d(\mathcal{P})$, the algorithm performs a regular match with s being the only constant in the *working-set* (Lines 2-5). Furthermore, if there are *started and unrelated* situations, we perform matches with s and each combination of them (Lines 6-8). This accounts for configurations as seen in Figure 4.2. All finished situations are migrated from the separate to the regular buffer (Lines 9-11) and if $s.te \in t_d(\mathcal{P})$, the matching process is triggered. This time with combinations of s and all *started and related* situations (Lines 15-16), further combined with all *started and unrelated* situations (Lines 17-18), which fuses the avoidance of duplicate results and false positives. An example for this case is shown in Figure 4.3. Note that, the actual constraint-checking among the created combinations is performed by the call to *performMatch* (Algorithm 3), since it is aware of pre-set situations in the *working-set*. As we will show

Algorithm 4: Low-Latency MatcherUpdate

Input: S_f, S_s : sets of finished/started, t : current time

```

1 purgeBuffers(t);
2 foreach  $s \in S_s$  do
3   startedBuffer.add(s);
4   if matchOnStart(s) then
5     performMatch( {s}, 0 );
6      $U \leftarrow$  getUnrelatedStarted(s);
7     foreach  $u \in \text{powerset}(U) \setminus \emptyset$  do
8       | performMatch(  $u \cup \{s\}$ , 0 );
9 foreach  $s \in S_f$  do
10  startedBuffer.remove(s);
11  addToBuffer(s);
12  if matchOnEnd(s) then
13     $R \leftarrow$  getRelatedStarted(s);
14     $U \leftarrow$  getUnrelatedStarted(s);
15    foreach  $r \in \text{powerset}(R) \setminus \emptyset$  do
16      | performMatch(  $r \cup \{s\}$ , 0 );
17      foreach  $u \in \text{powerset}(U) \setminus \emptyset$  do
18        | performMatch(  $r \cup u \cup \{s\}$ , 0 );
```

in section 6, the extensive building of combinations has only minimal impact on the runtime-performance, because it shifts load from joining to the update algorithm and does not introduce additional computation steps.

Duration constraints on situations are incorporated into low latency-matching with only a few modifications: First, if a maximum duration constraint is defined (regardless of a possibly specified minimum duration), the corresponding situation must not be included in the matching process until its end is known – and the constraint is fulfilled. Hence, these situations are excluded from the set of started situations (S_s) and if their start timestamp is in $t_d(\mathcal{P})$, matching is deferred to their end timestamp. Second, if a minimum but no maximum duration is defined, the inclusion into the set of started situations is deferred until the constraint is satisfied. This possibly implies the inclusion of its deferred start timestamp (\overline{ts}) into $t_d(\mathcal{P})$. As an example, consider the pattern A during B and the following order of timestamps: $B.ts < A.ts < A.te < B.\overline{ts} < B.te$. This match can not be detected at A.te, because B's duration does not exceed the lower bound at this point. Hence $B.\overline{ts}$ requires a matcher invocation.

To handle *prefix groups* the restriction that two *started and explicitly related* situations must not be matched is relaxed. That is, matching is performed if the corresponding *temporal constraint* contains one or more *prefix groups*. However, for still being able to omit false positives, the matcher must distinguish between *prefix group* and regular detection. Technically this means splitting the *temporal constraint* into two disjoint sets (one containing all *temporal relations* forming a *prefix group* and another one holding the remaining relations) and use the first set, when matching on a situation's start and the second one on its end.

5.4 Computing the Evaluation Order

The *matcher component* maps the problem of temporal pattern matching to a multi-way join between situation buffers. Like multi-join processing in traditional relational database systems,

Relation	before	during	overlaps	starts, finishes, meets	equal
Selectivity	0.445	0.03	0.01	0.0049	0.0006

Table 3: Initial estimates for the selectivity of temporal relations. Mirror relations are equivalent.

the performance of joining heavily depends on the order in which the join operations are executed. In this section, we discuss how the *matcher*'s evaluation order is computed and present the cost-model used during this process.

Analogous to classical join processing we implemented an optimizer that enumerates possible execution plans, computes the expected computational costs for each of them and suggests the most efficient plan for execution. We do not provide multiple implementations of the join operator, so that enumerating possible plans reduces to the enumeration of possible evaluation orders. To further reduce the number of plans to consider, we exclude orderings joining a situation buffer without an applicable *temporal constraint*. In other words: Plans involving the calculation of a cross product are omitted.

According to Equation 2, estimating the costs for a given plan boils down to estimating the size of intermediate results:

$$|R_i| := \begin{cases} |B_1| & \text{if } i = 1 \\ |R_{i-1}| \cdot |B_i| \cdot s_i & \text{otherwise} \end{cases} \quad (3)$$

s_i denotes the selectivity of the applicable *temporal constraints* in step i (C_i), which can be composed from the selectivities of the contained *temporal relations* as follows:

$$s_i := \prod_{C \in C_i} \left(\sum_{R \in C} s_R \right) \quad (4)$$

When a query is initially deployed into the system, the situation buffers are empty and we have no estimation on the selectivity of the *temporal constraints*. Hence, we initially assume the selectivities depicted in Table 3. These values are backed by the following back-of-the-envelope calculation: The combined selectivity of all possible relations should be 100%. Assuming equal sized buffers and an equal temporal distribution of the situations, the selectivity of a before relation will be around 50%. For during, the number of results is limited by the maximum of both buffer sizes, because a situation A can happen during at most one other situation (B), but B may contain multiple A situations. All other *temporal relations* define a 1:1 relationship, which limits the worst case to the minimum of both buffer sizes. As seen in Table 3, we additionally separate the last case by the number of stated equalities. Note that even though this is an initial estimate, the resulting plans prove to work well in most cases (cf. section 6.4.2).

5.4.1 Adaptivity. Once a query is deployed in a CEP-system, it is typically active for a long time. Hence, more important than the quality of an initial execution plan is the ability to tune this plan and adapt it to changing workloads. To do so, we keep track of the buffer sizes and selectivities imposed by *temporal constraints* during execution. The buffer-sizes are available at any point in time and at no cost, since they are tracked by the underlying data structure. However, to smooth out (potential) spikes, we monitor the buffer size using an *exponential moving average*, which is adjusted after each call to the *matcher*'s update method as follows:

$$EMA_i = \alpha * |B_i| + (1 - \alpha) * EMA_{i-1}$$

EMA_i holds after the i -th update. $|B_i|$ denotes the size of the considered buffer at update i and the *smoothing factor* $\alpha \in (0, 1)$ determines how much weight is given to previous values. For example, a value close to 1 assigns almost no weight to older values, while a value close to 0 decreases the influence of new values. The selectivities of the *temporal constraints* are also managed with EMAs using one EMA-value per constraint.

To check if a re-computation of the evaluation order is required, the active plan stores a snapshot of the statistics it is based on. After each update, we compare them to the current values and if any of them differs by more than the defined threshold (t), we trigger a re-computation.

Finally, if a migration is required, we are able to migrate to the new plan between any two invocations of the *matcher* component. Because the *matcher* does not store any intermediate results, but solely relies on the situation buffers this switch comes without any additional migration costs. As we will show in section 6.4.2, the total costs for adaptivity are negligible.

6 EXPERIMENTAL EVALUATION

In this section we present the results from our experimental evaluation of *TPStream*². First, we study *TPStream*'s evaluation performance in comparison to *ISEQ* and point based CEP systems. Then, we analyze the latency improvement of our approach in comparison to *ISEQ*. Finally, we prove the validity of our optimization techniques.

6.1 Setup

All experiments were conducted on a workstation equipped with an Intel i7-2600 3.4 GHz processor and 8GB of memory, running a Debian Linux (kernel version 4.11.11-1). The results presented for each experiment are averaged values from a total of 10 runs, whereby every run was preceded by a warm-up phase of evaluating at least 100,000 events before the measurement was started.

The main goal of this section is to compare *TPStream*'s processing performance and our low latency approach to the state-of-the-art solution for temporal pattern matching (*ISEQ*). There is no publicly available implementation of *ISEQ*, so we implemented it based on the available description in [20]. As required by the design of *ISEQ*, the input consists of interval streams ordered by endpoint. These streams are again generated with our *deriver* component.

In order to provide a comparison with point based systems, we also included CEP-solutions from the open-source community (Esper³ 6.0.1) and academia (SASE+⁴), when applicable. While Esper is a production ready CEP system, highly optimized for efficient query execution, SASE+ is one of the most popular CEP languages in the research community and served as foundation for the *ISEQ* operator. The rich query language of Esper allowed us to express both straw man's approaches as sketched in the introduction. We refer to the first approach (2 phase pattern matching) with *Esper-1* and the low latency approach is denoted as *Esper-2*. Because the SASE+ implementation does not feature chaining of queries, we only implemented the low-latency approach. *TPStream* and all its competitors are implemented in the JAVA programming language, whereby *TPStream* and *ISEQ* are based on JEPC [19] – an event processing middleware. We used

²Datasets and source code available at <http://uni-marburg.de/oaCPk>

³<http://www.espertech.com>

⁴<https://github.com/haopeng/sase>

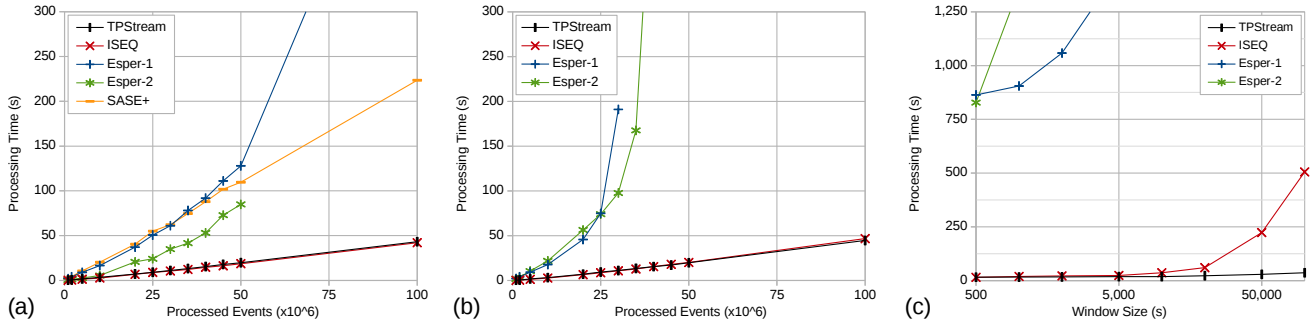


Figure 5: Processing time for aggressive driver detection as a function of the input size: (a) simplified pattern, (b) full pattern and processing time for disconnected pattern detection as a function of the window size (c)

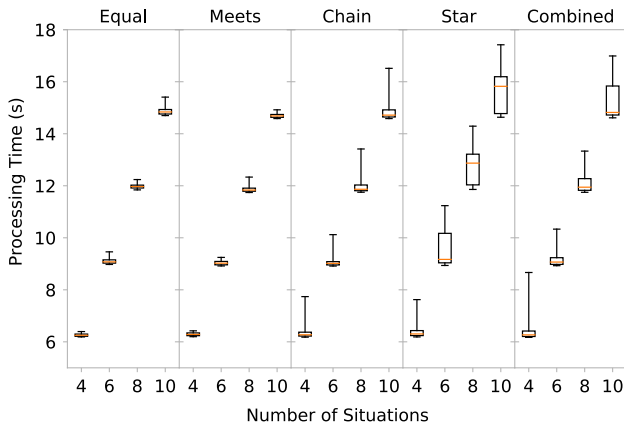


Figure 6: Processing time for various query patterns

Oracle JDK 1.8.0.144 compile the systems and ran all experiments on that JVM with 6GB of heap space.

During the evaluation two data sources were used. The first source comprises trip data generated with the Linear Road Benchmark [4]. Besides other attributes, each event consists of a unique car id, its location, the current speed and acceleration. We generated data simulating 5 hours of traffic on a single expressway with 1000 active cars per hour. Each active car reports its state every second, leading to 887 million events (36 GB of data). The second source is a random event generator, tuned to pose high load on the system. It generates event streams with a configurable number of boolean attributes, each representing a single situation stream. The generated situations last between 10 and 100 seconds, while the gaps between two consecutive situations span 10 to 50 seconds (both uniformly distributed). Events are generated with a frequency of 1Hz, so that for a situation lasting n seconds, the corresponding attribute’s value is true for exactly n consecutive events.

Independent of the data-source, we used a single thread for both, reading/generating the data and evaluating the query. For each experiment, we measured the reading/generation time upfront and removed it from the presented results. The most important parameters throughout all experiments are as follows:

Event Rate The rate (events/s) with which events are pushed into the systems.

Window Size The size of the time window (s) during which a pattern must occur completely.

Event Count The total number of events to process.

6.2 Processing Time

This set of experiments compares the processing performance of *TPStream* with its competitors using various queries and parameter settings. The events were pushed into the system at the maximum possible rate and we used the processing time as main measure.

6.2.1 Aggressive Drivers. We injected different fractions (1M to 100M events) of the Linear Road dataset into the system and executed the example query of Listing 1 (without duration constraints). The thresholds for speeding, acceleration and deceleration were the 99th, 90th and 90th percentiles for the speed and positive/negative acceleration values of a 50M event sample. Besides chaining of queries, the SASE+ implementation also lacks support for disjunctions. Nevertheless, to include SASE+ in this experiment, we also evaluated a simplified query version which restricts the used *temporal relations* to meets and overlaps.

The results of this experiment are shown in Figure 5 (a – simplified pattern, b – full pattern). The x-axis shows the number of processed events, the processing time is shown on the y-axis. *TPStream* and *ISEQ* are head to head and their processing times increase linearly with the number of processed events. Further, they are insensitive to alternatives, resulting in almost identical processing times for both query variants. *TPStream* was not able to outperform *ISEQ* in this experiment, because in the given pattern all situations overlap which in turn allows to break the buffer scan early. Esper benefits from the simplified version of the pattern, but its evaluation performance is inferior to *TPStream* and *ISEQ* (up to 30x for the full query and 15x for the simplified version). When evaluating the full query, Esper hit the memory limit of 6GB and the system crashed if more than 30M (Esper-1) and 40M (Esper-2) events were processed. For the simplified version, the processing time of Esper-1 increases drastically when processing more than 50M events – Esper-2 runs out of memory and crashes. SASE+ managed the evaluation, but was clearly outperformed by *TPStream* and *ISEQ*.

6.2.2 Disconnected Pattern. The second experiment compares processing time and memory consumption of the systems using a pattern with high selectivity: A before B overlaps C. The difference to the first experiment is, that each A situation may be related to many B overlaps C sub-matches instead of contributing to at most one match. Hence, we expected the number of results and consequently the processing time/memory consumption to depend on the size of the configured time window. We injected 300M synthetic events into the systems and executed

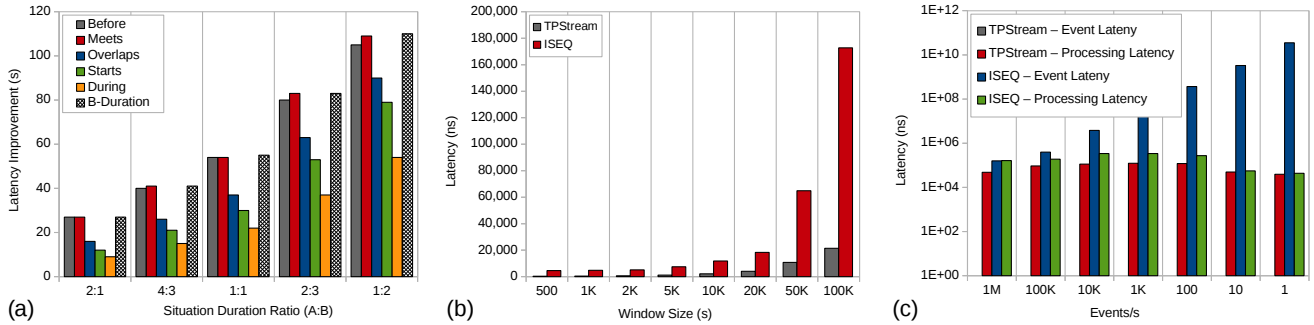


Figure 7: (a) application time latency gain per temporal relation and comparison of result latency (b) under maximum possible throughput as a function of the window size, (c) under varying event rates with a fixed size window

the query with window sizes varying from 500s (8:20 minutes) to 100,000s (slightly more than one day).

Figure 5 (c) shows the processing time of all systems as a function of the window size (note the log-scale). In this experiment, *TPStream* is able to outperform *ISEQ* by a factor of 14 using a window of 100,000s. This is because *ISEQ* does not make use of the order on the situations' start timestamp and requires additional computational steps during result construction and buffer pruning. *SASE+* did not finish this experiment in a reasonable time for none of the window sizes and *Esper* barely managed two window sizes up to 20,000s. To measure the average memory consumption, we monitored the used heap space with a frequency of 20Hz during each run and averaged these values. Both, *TPStream* and *ISEQ* require only very little additional memory for increased window sizes: *TPStream* 911 - 1018 MB, *ISEQ* 903 - 1027 MB. *Esper* stays stable at 1 GB up to a window size of 10,000s but afterwards suffers from buffering single events rather than a compact representation like situations. For the last evaluable query (20,000s window) *Esper* already consumed 1,7 GB of memory.

6.2.3 Query Patterns. To give a comprehensive overview of *TPStream*'s processing performance, we evaluated 5 different query patterns and varied the number of situation streams from 4 to 10. Queries 1-3 (**Equal**, **Meets**, **Chain**) are of the form $S_1 \oplus_1 S_2 \oplus_2 \dots \oplus_{n-1} S_n$, with $\oplus_i = \text{equals}$, $\oplus_i = \text{meets}$ and \oplus_i a randomly drawn temporal relation, respectively. In query 4 (**Star**), S_1 is connected with every other situation, via a random temporal relation (i.e. $S_1 \oplus_1 S_2, S_1 \oplus_2 S_3, \dots, S_1 \oplus_{n-1} S_n$). Query 5 (**Combined**) combines the two generic patterns by connecting the first $n/2$ situations via the **Chain** pattern and the remaining situations according to the **Star** pattern. Each query-type was executed 100 times, using 50M synthetic events and a window size of 2,000s.

The box plots in Figure 6 provide the median as well as the 25th and 75th percentiles of the processing time. For all query types, the median processing time increases linearly with the number of situations. The generic **Chain** pattern incurs higher maximum values than **Equal** and **Meets**, because the possible temporal relations include before, which is highly selective. This forces the matcher to build many partial results – especially if three or more consecutive situations are in a before relationship. **Star** queries are more sensitive to the concrete pattern instance, because in the worst case every situation triggers the matching process. This effect can also be observed for the **Combined** pattern, but to a smaller degree, because only half of the situations are connected via a **Star** pattern.

6.3 Low Latency

This set of experiments compares the result latency of our approach with the state-of-the-art solution for temporal pattern matching, *ISEQ*.

6.3.1 Application Time. At first, we measure the latency improvement of *TPStream* compared to *ISEQ* in terms of application time. That is, we compare the timestamps of the events that produced a result in both approaches and calculate their difference. We evaluated each temporal relation independently using two synthetic situation streams (A, B). We varied the average duration ratio from 2:1 to 1:2, keeping A's average duration fixed at 55 seconds. Note that the window size has no impact here (as long as it is not too small to hold a match), so it was set to 1,000s.

Figure 7 (a) shows the average latency improvements per temporal relation. For sequential relations (before, meets), the gain in latency is equal to the average duration of B situations, because matches are detected at B.ts. For the remaining relations, the detection time is A.te and the average improvement depends on the concrete temporal relation. In the worst case (during) this is B.duration/2. Note that, equals and finishes were not included, because no latency improvements can be achieved.

6.3.2 Wall Clock Latency. We conducted two experiments, showing that *TPStream*'s processing techniques significantly reduce the result latency in terms of wall clock time which is a critical aspect in a streaming scenario. Therefore, we repeat the experiment from section 6.2.2 twice: first, we measure the time passed between the arrival of the first event that could produce a result and the receipt of that result. We varied the window size and pushed events with the maximum possible rate. For the second experiment we fixed the window size at 100,000s and varied the event rate from 1M to 1 events/s. This time, we split the measured latency in (i) processing latency: the time passed between arrival of the event that triggered the result and the actual receipt of that result and (ii) event latency: the time passed between arrival of the first event that could trigger the result and the arrival of the event that actually triggered that result.

The results are shown in Figure 7 (b,c). Both figures show the average latency per result (y-axis, note the log-scale for c). While (b) shows, that *TPStream*'s evaluation techniques provide latency savings through reduced processing time, (c) highlights the savings achieved with our low-latency matcher. Especially when the rate is in sync with application time (1 event/s), the event latency of *ISEQ* dominates the processing latency and almost reaches the application time savings (~35s, cf. Figure 7 a, 1:1, overlaps), while *TPStream* introduces no event latency at all.

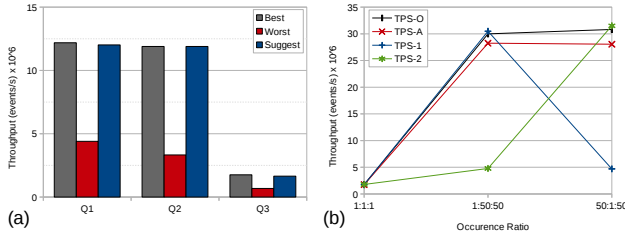


Figure 8: (a) Quality of the initial plans for Q1 – Q3, (b) Throughput comparison: dynamic plan adaption vs. best initial plans

6.4 Plan Quality & Adaption

Finally, we evaluate the optimization techniques presented in section 5.4. Like in section 6.2, events were pushed with the maximum possible rate.

6.4.1 Initial Plan Quality. To evaluate the quality of the generated initial plans, we used the following queries on three situation streams: **Q1**: A overlaps B AND A overlaps C AND B starts C, **Q2**: A overlaps B AND A before C AND B overlaps C and **Q3**: A before B AND A before C AND B before C. For each query, we generated all 6 valid plans and measured the throughput (processed events/s) by evaluating synthetic events with a window size of 5,000s.

Figure 8 (a) shows the results for the best, worst and suggested plans and clearly confirms our approach. For queries **Q1** and **Q2** the best plan was suggested. The initial plan for **Q3** was $C \rightarrow B \rightarrow A$ even though the estimated costs for $C \rightarrow A \rightarrow B$ are the same. The experiments show, that $C \rightarrow A \rightarrow B$ would have been a slightly better choice, but the difference is negligible.

6.4.2 Dynamic Plan Adaption. To analyze the plan adaption capabilities of *TPStream*, we executed **Q3** again and processed 300M events. The occurrence ratio of situations A, B and C changed from 1:1:1 to 1:50:50 after 100M events and finally to 50:1:50 after 200M events. The window size, smoothing-factor (α) and threshold for plan migration (t) were set to 10,000s, 0.01 and 0.2, respectively. Besides the adaptive implementation (TPS-A), we ran the experiment with both best initial plans $C \rightarrow B \rightarrow A$ (TPS-1), $C \rightarrow A \rightarrow B$ (TPS-2), and an implementation, doing a hard coded switch to the best plan exactly when the characteristics of the stream changes (TPS-O).

Figure 8 (b) shows the throughput for all four configurations and the three different stream-characteristics: TPS-1 and TPS-2 both have drawbacks in either one of the skewed phases, while our adaptive approach is very close to the optimal solution TPS-O (suffering slightly from dynamic adaption). However, the total runtime of TPS-O (63,523ms) compared to TPS-A (64,612ms) reveals only a negligible overhead of 1,089ms (less than 2%) for plan adaption.

7 CONCLUSION

We presented *TPStream*, a novel event processing operator for detecting complex *temporal patterns* among event streams. We enabled *TPStream* to derive lasting situations directly from streams of events and developed new techniques for detecting *temporal patterns* at the earliest possible point in time. Furthermore, we demonstrated low-cost adaptive approaches suitable for a streaming scenario. We proved the potential of *TPStream* by comparing it to industrial and academic solutions for CEP in experiments.

Since research on situations in CEP is scarce, we focused our efforts on presenting a fundamental solution suited for this scenario and equipped it with the capabilities to handle the adaptive, low-latency nature of stream processing. For future work, we intend to extend *TPStream* to tackle out-of-order arrivals [7, 21] and parallel processing [24, 26].

ACKNOWLEDGMENTS

This work has been supported by the German Research Foundation (DFG) under grant no. SE 553/9-1.

REFERENCES

- [1] Mohamed H. Ali and others. 2009. Microsoft CEP Server and Online Behavioral Targeting. *Proc. of the VLDB Endowment* 2, 2 (2009), 1558–1561.
- [2] James F. Allen. 1983. Maintaining knowledge about temporal intervals. *Comm. of the ACM* 26, 11 (1983), 832–843.
- [3] H.-Jürgen Appelrath and others. 2012. Odysseus: a highly customizable framework for creating efficient event stream management systems. In *DEBS'12*. 367–368.
- [4] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, and Anurag Maskey. 2004. Linear road: a stream data management benchmark. In *VLDB'04*. 480–491.
- [5] Ron Avnur and Joseph M. Hellerstein. 2000. Eddies: Continuously Adaptive Query Processing. In *SIGMOD'00*. 261–272.
- [6] Shivnath Babu, Rajeev Motwani, Kamesh Munagala, Itaru Nishizawa, and Jennifer Widom. 2004. Adaptive ordering of pipelined stream filters. In *SIGMOD'04*. ACM, 407–418.
- [7] Badrish Chandramouli, Jonathan Goldstein, and David Maier. 2010. High-performance dynamic pattern matching over disordered streams. *Proc. of the VLDB Endowment* 3, 1 (2010), 220–231.
- [8] Alan J. Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, and Walker M. White. 2007. Cayuga: A General Purpose Event Monitoring System. In *CIDR'07*. 412–422.
- [9] Yanlei Diao, Neil Immerman, and Daniel Gyllstrom. 2007. *Sase+ : An agile language for kleene closure over event streams*. Technical Report. University of Massachusetts.
- [10] Martin Erwig. 2004. Toward Spatio-Temporal Patterns. In *Spatio-Temporal Databases: Flexible Querying and Reasoning*. Springer Berlin Heidelberg, 29–53.
- [11] Opher Etzion, Fabiana Fournier, Inna Skarbovsky, and Barbara von Halle. 2016. A model driven approach for event processing applications. In *DEBS'16*. 81–92.
- [12] Ioannis Flouris, Nikos Giatrakos, Antonios Deligiannakis, Minos Garofalakis, Michael Kamp, and Michael Mock. 2016. Issues in complex event processing: Status and prospects in the Big Data era. *Journal of Systems and Software* (2016).
- [13] Dengfeng Gao, Christian S. Jensen, Richard T. Snodgrass, and Michael D. Soo. 2005. Join operations in temporal databases. *VLDB Journal* 14, 1 (2005), 2–29.
- [14] Thanaa M. Ghanem, Walid G. Aref, and Ahmed K. Elmagarmid. 2006. Exploiting predicate-window semantics over data streams. *SIGMOD Record* 35, 1 (2006), 3–8.
- [15] Lukasz Golab and M Tamer Özsu. 2003. Processing sliding window multi-joins in continuous queries over data streams. In *VLDB'03*. 500–511.
- [16] Lukasz Golab and M. Tamer Özsu. 2005. Update-pattern-aware modeling and processing of continuous queries. In *SIGMOD'05*. 658–669.
- [17] Michael Grossniklaus, David Maier, James Miller, Sharmadha Moorthy, and Kristin Tuft. 2016. Frames: Data-driven windows. In *DEBS'16*. 13–24.
- [18] Annika Hinze and Agnès Voisard. 2015. EVA: An event algebra supporting complex event specification. *Information Systems* 48 (2015), 1–25.
- [19] Bastian Hoßbach, Nikolaus Glombiewski, Andreas Morgen, Franz Ritter, and Bernhard Seeger. 2013. JEPC: The Java Event Processing Connectivity. *Datenbank-Spektrum* 13, 3 (2013), 167–178.
- [20] Ming Li, Murali Mani, Elke A. Rundensteiner, and Tao Lin. 2011. Complex event pattern detection over streams with interval-based temporal semantics. In *DEBS'11*. 291–302.
- [21] Mo Liu, Ming Li, Denis Golovnya, Elke A. Rundensteiner, and Kajal Claypool. 2009. Sequence Pattern Query Processing over Out-of-Order Event Streams. In *ICDE'09*. 784–795.
- [22] Yuan Mei and Samuel Madden. 2009. ZStream: a cost-based query processor for adaptively detecting composite events. In *SIGMOD'09*. 193–206.
- [23] Olga Poppe, Chuan Lei, Elke A. Rundensteiner, and Daniel J. Dougherty. 2016. Context-Aware Event Stream Analytics. In *EDBT 2016*. 413–424.
- [24] Medhabi Ray, Chuan Lei, and Elke A Rundensteiner. 2016. Scalable pattern sharing on event streams. In *SIGMOD'16*. 495–510.
- [25] Mahmoud Attia Sakr and Ralf Hartmut Güting. 2011. Spatiotemporal pattern queries. *Geoinformatica* 15, 3 (2011), 497–540.
- [26] Nicholas Poul Schultz-Møller, Matteo Migliavacca, and Peter Pietzuch. 2009. Distributed complex event processing with query rewriting. In *DEBS'09*. 1–12.
- [27] Haopeng Zhang, Yanlei Diao, and Neil Immerman. 2014. On complexity and optimization of expensive queries in complex event processing. In *SIGMOD'14*. ACM Press, 217–228.

QUASII: QUery-Aware Spatial Incremental Index

Mirjana Pavlovic
EPFL
mirjana.pavlovic@epfl.ch

Thomas Heinis
Imperial College
t.heinis@imperial.ac.uk

Darius Sidlauskas
EPFL
darius.sidlauskas@epfl.ch

Anastasia Ailamaki
EPFL & RAW Labs SA
anastasia.ailamaki@epfl.ch

ABSTRACT

With large-scale simulations of increasingly detailed models and improvement of data acquisition technologies, massive amounts of data are easily and quickly created and collected. Traditional systems require indexes to be built before analytic queries can be executed efficiently. Such an indexing step requires substantial computing resources and introduces a considerable and growing data-to-insight gap where scientists need to wait before they can perform any analysis. Moreover, scientists often only use a small fraction of the data — the parts containing interesting phenomena — and indexing it fully does not always pay off.

In this paper we develop a novel incremental index for the exploration of spatial data. Our approach, QUASII, builds a data-oriented index as a side-effect of query execution. QUASII distributes the cost of indexing across all queries, while building the index structure only for the subset of data queried. It reduces data-to-insight time and curbs the cost of incremental indexing by gradually and partially sorting the data, while producing a data-oriented hierarchical structure at the same time. As our experiments show, QUASII reduces the data-to-insight time by up to a factor of 11.4x, while its performance converges to that of the state-of-the-art static indexes.

1 INTRODUCTION

The advances in data acquisition technologies and supercomputing for large-scale simulations rapidly increase the amounts of spatial data generated and collected. For instance, in the Human Brain Project (HBP) [27], neuroscientists build spatial models of the brain which will ultimately feature 10^{11} neurons [42], each reconstructed with thousands of 3d cylinders. NASA released 500 TB of earth observation data generated through remote sensing [30], while the Dutch government released point cloud data with 640 billion points [31] acquired through airborne scanning. Similarly, volunteers generate large amounts of spatial data through services such as OpenStreetMap [33]. Given these massive and growing amounts of spatial data, algorithms to query them efficiently are crucial.

Previous research has proposed many techniques [11, 26, 42] for the fast and scalable querying of spatial datasets. Existing approaches, however, have two major drawbacks. First, they require a time-consuming step to build indexes before they can be used. This pre-processing step significantly delays the analyses: indexing a model in the HBP, for example, can take several hours [42]. With increasing dataset size, the data-to-insight time grows as well. Second, scientists frequently only analyse a small fraction of the data [1, 8]. In the HBP, for example, a scientist builds a

model of the brain but after a few queries may determine that it is not biorealistic (e.g., density in certain areas does not agree with measurements) and stops the analysis. Given the small number of queries executed, the overhead of indexing the entire model cannot be fully amortized.

The problems of delayed analysis (due to prior indexing) and the impossibility to amortize indexing cost (due to too few queries) are not exclusive to spatial data management. Database research has proposed incremental indexes for relational data (e.g., cracking [18] and adaptive merging [14]) and for time-series [45]. The core idea is to incrementally index only the parts of the data queried, spreading the cost of indexing over the first few queries. The major data-to-insight bottleneck is thus eliminated, i.e., queries are answered as soon as data is available (albeit the first queries run slower, as no index is initially available).

In this paper, we develop an incremental indexing approach for spatial data in main memory, with the aim of reducing data-to-insight time, as well as achieving performance comparable to traditional spatial indexes (after enough queries are executed). As no current incremental indexing approach for main memory exists, we demonstrate the limitations of applying current options to incrementally index spatial data. As we show, using the concepts for incrementally indexing one-dimensional data [18] to index three-dimensional data does not significantly reduce data-to-insight time, as the major bulk of work still has to be done for the first query. Adapting Space Odyssey [35], an incremental index for exploratory analyses of multiple spatial datasets on disk, to main memory leads to excessive reorganization of the data. As a consequence, a static index (including pre-processing cost) quickly outperforms the proposed incremental solution, in terms of total execution time.

We thus develop a QUery-Aware Spatial Incremental Index - QUASII: a novel data-oriented, query-driven incremental indexing approach. QUASII substantially reduces data-to-insight time and keeps the cost of incremental strategy low, by gradually and partially sorting the spatial objects considering all dimensions. QUASII thus distributes the cost of indexing across all queries, while preserving spatial proximity and producing a data-oriented style partitioning — which typically entails an expensive pre-processing step in the static setting. Finally, being data-oriented, it executes queries efficiently, as it adjusts to the distribution of the data, while avoiding data replication.

Our experiments show that QUASII substantially accelerates the exploratory analysis of spatial data in main memory by reducing the data-to-insight time by up to 11.4x, while achieving the query performance of current algorithms for spatial indexing. Static algorithms are not able to amortize their building cost over QUASII even after 10000 queries.

To our knowledge we are the first to develop and analyze incremental indexing for spatial data. Our contributions are:

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

- We demonstrate the challenges of adapting and using known incremental indexing [18, 35] to spatial data in main memory. We use the resulting approaches as motivation and baseline.
- We develop *QUASII*, an incremental approach that significantly reduces the data-to-insight time, while achieving the query performance of state-of-the-art spatial indexes.
- We experimentally analyse *QUASII*'s performance and the number of queries it needs to reach the performance of its static counterparts.

The remainder of the paper is structured as follows. We define the problem in Section 2 and motivate it in Section 3. We then describe *QUASII* in Sections 4 and 5 and experimentally evaluate it in Section 6. Section 7 gives an overview of related work before we conclude in Section 8.

2 PROBLEM DEFINITION

Our work is driven by the need for the exploratory analysis of spatial datasets through querying. The queries executed are ad hoc, i.e., the next query is only known after the results of the first query are analyzed, and they thus cannot be batched and executed with only one sequential read of the dataset.

Example Application. In the Human Brain Project, neuroscientists build spatial models of the brain [27]. Already now the models are so detailed that to simulate a neocortical volume of only 0.29 mm^3 supercomputers are needed [28].

Once the part of a model is built, neuroscientists need to validate it by choosing a subset of its regions at random and inspecting them. Each region is queried with several spatially close queries and the query results are used to verify the composition, density and other metrics agree with the real brain. The results of these analyses are crucial to determine whether or not the model can be simulated or should be abandoned (subsequently building a new one using a different configuration). Scientists currently only have two fundamentally different options: index all data a priori and execute queries with the index or scan all data each time to answer a query. Not knowing a priori how many queries will be executed (and if indexing can be amortized) makes it difficult to decide.

Data. We consider spatially extended (volumetric) objects enclosed by a minimum bounding box (MBB). In a three-dimensional ($3d$) setting, each MBB b is defined by two $3d$ points $lower(b)$ and $upper(b)$ corresponding to lower and upper coordinate at each dimension ($lower(b) = (x_l, y_l, z_l)$ and $upper(b) = (x_u, y_u, z_u)$) [11].

Queries. We focus on range (window) queries as they are broadly used in many applications and are also the building block for many other spatial queries (e.g., k -nearest neighbor queries [22]). Each query is a $3d$ box specified by two $3d$ points, e.g., (q_l, q_u) . Given a query q , all objects with their bounding box b intersecting with q , i.e., where $b \cap q \neq \emptyset$, are in the result.

Setting. We assume that all data and necessary index structures fit in main memory. We consider a static setting, i.e., all raw data is available before querying.

3 MOTIVATION

No current incremental indexing approach can index spatial data in main memory. Research has developed incremental indexing for relational, one-dimensional data in main memory, i.e., cracking [18] and for spatial data on disk [35]. In the following we extend the former [18] to the spatial domain and adapt the latter [35] to use in main memory — to demonstrate the limitations

of these ideas in reducing the data-to-insight time and to motivate the need for a new approach.

3.1 Cracking for Spatial Data

Relational Cracking. Database cracking [16, 18, 19] incrementally builds an index as a byproduct of query execution in the context of main memory column-stores. The proposed techniques partially sort elements based on the query execution, essentially performing an incremental quick sort. In its simplest form, cracking [18] rearranges elements in an array according to the end points of the query range (q_l, q_u) : all values $< q_l$ are moved towards the beginning of the array, while values $> q_u$ are moved towards the end. With each query, the index becomes more refined until it is fully sorted and indexed.

SFCracker. Using this strategy to index spatial data is inherently challenging: spatial data has multiple dimensions and, unlike $1d$ data, no total order can be directly imposed on it. Therefore, to be able to use the strategy of cracking we transform data from the multi- to the one-dimensional domain. We perform this transformation using a space-filling curve (SFC) — a common approach to impose a total, $1d$ order on spatial objects.

A SFC maps data to $1d$ domain by visiting all the points in a d -dimensional grid exactly once; the order in which the objects are visited defines their order in $1d$ space. When mapping spatial data, it is crucial to consider SFCs that preserve proximity (such as Z-order [34] or the Hilbert curve [21]), so that data points close in multi-dimensional space remain close in $1d$ space [10, 29].

The resulting approach, *SFCracker*, incrementally sorts SFC codes based on the queried region. Both, data and queries are transformed to $1d$ space. The data transformation takes place in the first query, which makes it the most expensive one. Once the data is transformed, the queries perform cracking based on the $1d$ intervals obtained through the query transformation.

A naive query transformation to $1d$ space results in a substantial number of false positives (needed to be tested for intersection) because the transformed $1d$ range can be significantly larger than the original multi-dimensional range if only the *lower* and *upper* coordinates of the range query are considered. An example is shown in Figure 1: the curve segments in blue belong to the transformed range ($SFCcode_l, SFCcode_u$),

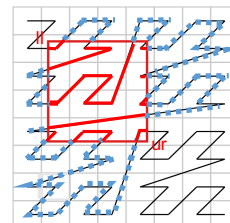


Figure 1: 1d transformation: overhead.

but they are outside of the original query range (in red). To reduce the overhead of false positives, we use a technique that partitions the curve into multiple sub-intervals each of which is fully contained in the original range [43]. Consequently, a range query is transformed into a number of intervals and the data is thus cracked multiple times per query, once for every interval.

Limitations. Cracking in the relational domain decreases data-to-insight time, distributing the cost of sorting over all queries with fairly low overhead and initialization cost. These benefits, however, decrease for datasets with a higher number of dimensions. First, the initial query is expensive as it maps all the objects from the multi- to the one-dimensional domain. Second, as opposed to relational data, a single query has to perform multiple expensive cracks to avoid performance penalties introduced with the transformation to $1D$ space. Consequently, spatial cracking still has a considerable data-to-insight time, along with an

expensive incremental strategy. We demonstrate these limitations experimentally in Section 6.3.

3.2 Disk-based Incremental Indexing in Main Memory

Disk-based Incremental Indexing. Space Odyssey [35] is a recently proposed incremental index for the exploration of spatial data. However, it tackles a different problem: Space Odyssey is designed for exploratory analyses of multiple spatial datasets. Without prior information, it incrementally indexes the datasets and adapts the physical layout of the data on disk for datasets frequently queried together. Although Space Odyssey addresses a different problem, we use its ideas related to incremental indexing and adapt them for use in main memory in *Mosaic*.

Mosaic. Mosaic incrementally builds an Octree [20] by dividing the space uniformly into eight partitions. Figure 2 depicts the indexing process (in 2d for clarity). For every query, Mosaic identifies the partitions that overlap with the query, splits them into eight partitions and reassigns their objects to the newly created partitions. Frequently queried areas in a dataset are indexed fully, whereas less frequently queried areas are coarser grained. The top-down strategy is thus beneficial for consecutive queries, as they can reuse the previous partitioning, independent of the workload pattern. However, data in frequently queried areas is re-partitioned multiple times.

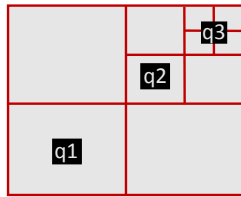


Figure 2: Mosaic: incremental strategy.

Limitations. Mosaic introduces significant overhead as the data in frequently queried areas is re-partitioned multiple times until it reaches its final configuration. Consequently, a static approach based on space-oriented partitioning, such as the uniform grid, outperforms quickly Mosaic in terms of total execution time (we provide more details in Section 6.3).

Mosaic additionally suffers from considering more objects than strictly necessary – a problem inherent in space-oriented partitioning and related to data assignment. For indexes based on space-oriented partitioning, objects can be assigned to cells with two strategies: replication and query extension. Replication assigns an object to all partitions that it overlaps with. As a consequence more objects need to be considered for intersection, the memory footprint increases and an expensive de-duplication step is needed. The alternative is to use query extension [40] which assigns an object to a cell based only on its center. This technique avoids object replication, however, it can considerably increase the number of objects necessary to be tested for intersection. More precisely, to ensure the correctness of the query result, it extends the query range by the maximum object extent. As a result, the area queried for is bigger than the initial query. Both strategies, replication and query extension, slow down query execution but, as we show in Section 6.2, replication is particularly expensive when working with volumetric spatial objects and we thus use query extension in Mosaic.

4 QUASII OVERVIEW

As discussed, an approach to incrementally index spatial data is not as straightforward as adapting known approaches. Besides the challenges, we also identify important design goals:

- (i) **minimal data-to-insight time:** the main requirement for incremental indexing is to enable instant access to the data, i.e., the first queries must not introduce undue overhead/processing;
- (ii) **efficient query performance:** the performance of frequently queried subsets of data should converge to that of the fully indexed approach (or better);
- (iii) **low cost incremental indexing:** indexing should introduce as little overhead as possible, i.e., its cumulative execution time should only exceed the one of static indexes after as many queries as possible (or not at all).

Given the design goals and our analyses, we develop QUery-Aware Spatial, Incremental Index, QUASII. QUASII is a data-oriented index, incrementally built as a side effect of query execution. It reduces data-to-insight time and curbs the cost of incremental indexing by gradually and partially sorting the data, while simultaneously producing a data-oriented hierarchical structure. It is based on a *nested reorganization strategy* which incrementally slices the space in each dimension and a *hierarchical, data-oriented structure* designed to accommodate the incremental indexing process and provide efficient query execution.

Overview. Figure 3a illustrates QUASII’s incremental strategy on a high level. Given range queries of the form $q = [q_l = (x_l, y_l, z_l), q_u = (x_u, y_u, z_u)]$, QUASII reorganizes the objects based on each query’s lower (q_l) and upper (q_u) coordinate by slicing each dimension and performing a nested reorganization. It first reorganizes objects on the x dimension, producing three x slices where the middle one contains the objects in the range $[x_l, x_u]$ given the query range in dimension x . Subsequently, it continues reorganizing the middle x slice on the y dimension, producing again three slices where the middle one contains objects in the range $[y_l, y_u]$. Finally, QUASII reorganizes the y slice on the z dimension producing the z slice which contains the query result. QUASII never performs a complete sort but reorganizes data locally, given the query’s boundaries.

The slices produced are organized in a hierarchical structure that incrementally forms the index. Figure 3b illustrates the structure of QUASII after the very first query (left) and after an arbitrary number of queries (right) are executed. QUASII forms a hierarchical structure with one level per dimension, i.e., the first (top), second, and third (bottom) levels correspond to slices at x , y , and z dimensions, respectively. The top level has the coarsest granularity as its objects are constrained with one dimension, while the bottom level is the most fine-grained since it is constrained by all dimensions. When executing the queries, QUASII traverses the structure depth-first, performing additional refinements when necessary, as we discuss later in Algorithm 1.

Nested Reorganization Strategy. The incremental strategy of QUASII is query-driven and data-oriented. Being query-driven, it reorganizes the minimal amount of data while executing queries. At the same time, being data-oriented, it achieves query efficiency as it adjusts to the data distribution, while avoiding replication. QUASII accomplishes both through its nested reorganization.

Data-oriented partitioning typically entails an expensive pre-processing step in the static setting as it preserves spatial proximity based on a strategy for ordering multi-dimensional objects. QUASII distributes the cost of this pre-processing across all queries by performing nested and partial reorganization. It reorganizes only a subset of data driven by queries, gradually curbing the amount of data partially sorted with every dimension. This strategy is inspired by the Sort-Tile-Recursive (STR)

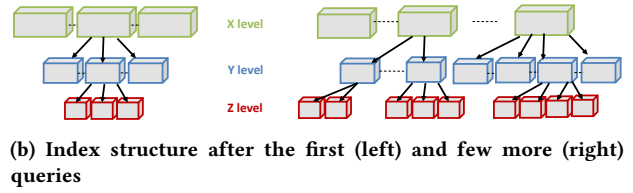
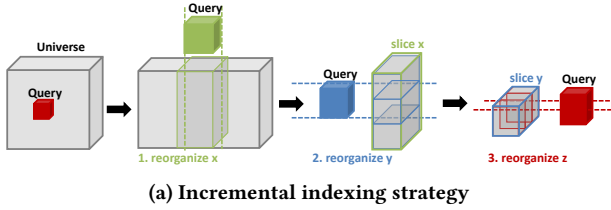


Figure 3: QUASII incremental indexing strategy and data structure.

R-tree bulkloading algorithm [26]. STR produces tiles that form leaf-level nodes for the R-Tree by recursively, fully sorting spatial objects in each dimension. More precisely, STR for $3d$ objects first sorts the spatial objects on the x -axis and partitions them in vertical tiles of equal size (i.e., the same number of objects). Then, within each x tile, it recursively applies the same strategy first considering y and then z dimension. This tiling strategy is particularly efficient as the resulting R-Tree has less overlap than other approaches [26]. By only performing partial reorganizations for the parts of the data that is actually queried, QUASII outputs partitions targeting these characteristics at lower cost (as opposed to complete sorts in STR).

Index Structure. QUASII’s index structure is designed to support an efficient incremental strategy with as little performance penalty as possible. Its hierarchical structure is designed to accommodate the reorganization strategy: each level corresponds to one (reorganization) dimension and each parent node is represented by its children in a nested form along the dimensions QUASII reorganizes data. We discuss the data structure and how it accommodates incremental indexing in more detail in Section 5.1.

Benefits. Ultimately, the design choices behind our approach enable us to achieve the goals we outlined. To reduce data-to-insight time (i), QUASII keeps data in the multi-dimensional, spatial domain. This avoids transforming all data at the very beginning which significantly hurts performance of the first query. Next, to achieve query efficiency (ii), QUASII uses data-oriented partitioning that preserves spatial proximity, adjusts to the distribution of data, and avoids object replication. Finally, to keep the cost of the incremental indexing low (iii), QUASII gradually and partially sorts the data using a nested reorganization strategy.

5 DATA STRUCTURE & QUERY PROCESSING

In the following, we explain the QUASII index structure and data organization before we proceed with discussing querying and incremental indexing algorithms.

Throughout this section, we refer to a $2d$ example given in Figure 4. It depicts a dataset $D = \{o_0, \dots, o_9\}$ of ten (gray) rectangular spatial objects. All subfigures have three main parts: the top part shows a $2d$ view of the dataset D and how the space is conceptually “sliced” by QUASII, the middle (“Data array”) depicts how (raw) data objects are re-organized in main memory, and the bottom shows QUASII’s hierarchical data structure that is incrementally built. All x - and y -axis related slicing is marked in green and blue, respectively. Figure 4a) shows the initial state: the “slice-less” view of the data space with D objects and the very first query q_1 , the data array of spatial objects in an arbitrary initial order, and the data structure containing the initial slice, s_0 (capturing the entire dataset).

5.1 Data Structure

QUASII forms a d -level hierarchical structure, organized according to the number d of dimensions. Each level l has a one-to-one mapping to the corresponding dimension. That is, the first level ($l = 1$) represents slicing of data at x , the second level ($l = 2$) slices at y , and the third level ($l = 3$) slices at the z dimension. The top level always slices data objects at the coarsest granularity, while the bottom level is the most fine-grained. Each slice is described with four attributes: (i) its level, (ii) a minimum bounding box capturing all its objects, (iii) indices to the data array corresponding to the first and last entry of the objects that belong to the slice, and (iv) pointers to sub-slices refining the slice further on the subsequent dimension. In Figure 4, this corresponds to the four fields present in each node of the data structure (next to slice label, e.g., s_0): l , box , ids , and arrow pointers (when not null). In our two-dimensional view of the dataset, we mark $boxes$ with a solid line (in the corresponding color), while the slice cuts are marked as dashed lines.

Data-oriented Slicing. One of the main advantages of data-oriented partitioning is that each spatial object is always assigned to just one partition (slice). However, QUASII determines the slices in each dimension based on query ranges. Given volumetric spatial objects, objects can be sliced through and thus overlap with multiple slices. To overcome this problem, QUASII represents each object using only one of its coordinates and uses this coordinate to identify a slice where an object will be assigned to. In particular, during indexing, it uses each object’s lower coordinate (x_l, y_l, z_l). Being part of object’s MBB, this does not require any additional computation or storage¹. In Figure 4, this coordinate is marked as a black dot for all objects. Figure 4b illustrates slicing based on the very first query q_1 and its range $[2, 4]$ on the x -axis. Slicing at $x = 2$ and $x = 4$ results in three x -slices (s_1, s_2 , and s_3). While object o_6 overlaps two slices (s_2 and s_3), it is assigned to s_2 based on its lower coordinate (x_l). Note how the objects are re-organized in the data array and correspond to three partitions (slices) with coordinates $x < 2$, $2 \leq x \leq 4$, and $4 < x$. Accordingly, the data structure is updated with three new (more refined) slices replacing the initial (coarser) slice s_0 (capturing the whole dataset).

While QUASII assigns objects to slices based on their single (lower) coordinate, it records a minimum bounding box for each slice taking into account the actual spatial extent of the objects and thus ensures the correctness of the query result. This also results in slice representations (their MBBs) that are often much smaller but not necessarily within the originally sliced bounds. For example, s_1 contains only one object and thus has a very small MBB (i.e., its $box = o_2$), while the MBB of s_2 is b_2 and exceeds the original cut at $x = 4$ (Figure 4b). As we show later, this enables QUASII to discard many unnecessary slices during

¹The upper coordinate (x_u, y_u, z_u) or the object’s center (requires to be computed, though) can equally be used.

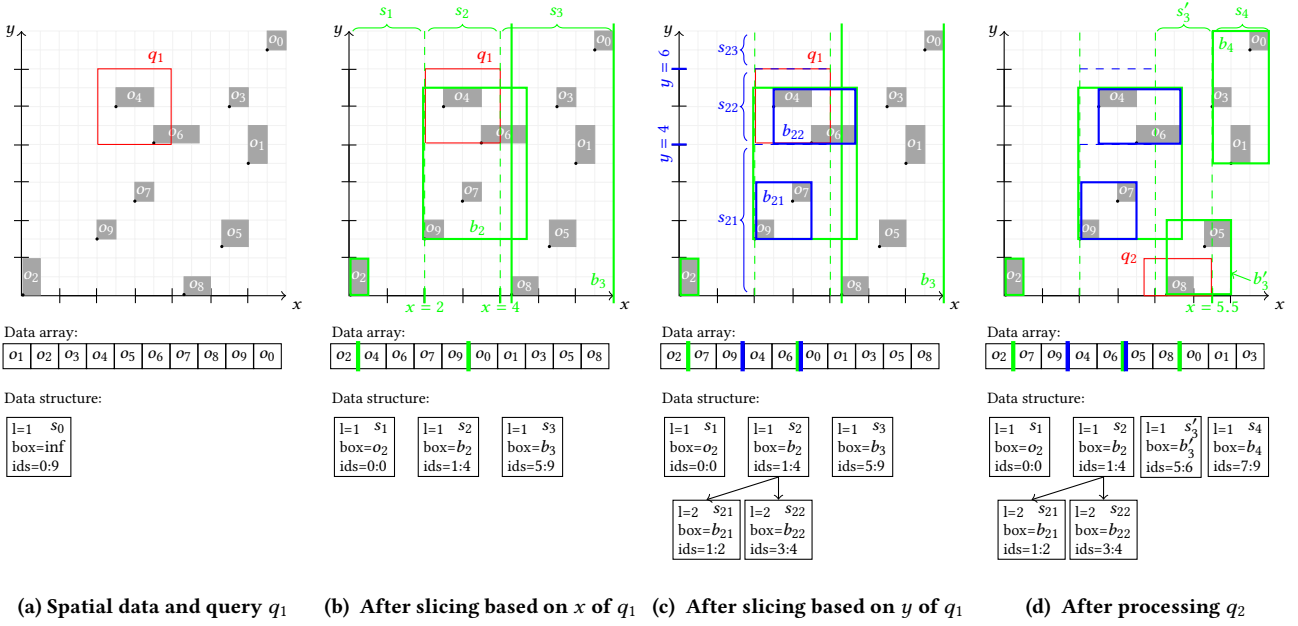


Figure 4: An example of query processing and incremental indexing in QUASII (configured with $\tau_x = 4$ and $\tau_y = 2$), given ten spatial objects (o_0 – o_9) and two range queries (q_1 and q_2).

query execution. To limit unnecessary computation (as a slice can be reorganized multiple times until it is fully refined), QUASII computes a full MBB only when a slice is completely refined. Otherwise, a slice is represented with an open-ended MBB, i.e., the MBB has bounds only on the dimension it has been sliced on.

Configuration. QUASII has only one configuration parameter, a size threshold τ , that determines the maximum number of objects in a slice at the finest level. That is, at the bottom level, whenever a slice s contains less or τ number of objects (i.e., $|s| \leq \tau$), it is considered to be fully refined. Intuitively, this is similar to setting a (leaf) node size in the R-Tree.

The sizes of the remaining $d-1$ levels are calculated as follows. Since QUASII performs data-oriented slicing, the total number of partitions required to satisfy threshold τ is $\lceil n/\tau \rceil$, where n is the total number of objects (i.e., $n = |D|$). Consequently, the number of times QUASII has to slice the data space across each dimension to produce $\lceil n/\tau \rceil$ partitions is equal to:

$$r = \lceil \sqrt[d]{n/\tau} \rceil \quad (1)$$

If we use τ_d to denote the slice threshold at the bottom level $l = d$ (i.e., $\tau_d = \tau$), then the maximum number of objects per slice for the remaining levels (up to the top) can be expressed recursively as $\tau_{d-1} = r \times \tau_d$. Note that r corresponds to the number of sub-slices (within a slice) at each index level.

Turning to our $2d$ example², after x -based slicing in Figure 4b, s_1 contains one object and thus is considered fully refined (i.e., $|s_1| = 1 \leq \tau_x$), while s_3 has five objects and may be refined in the future. Also note that s_3 stores an open-ended MBB ($s_3.box = b_3$).

The number of levels in QUASII is fixed and always equals to the dimensionality of the queried dataset. That is, it does not depend on the size of the dataset. Therefore, to accommodate the index growth (the index grows in breadth) and enable efficient query execution, QUASII keeps the children (within a slice) organized/sorted according to the level’s dimension. QUASII uses this order and the minimum bounding boxes (box) of each node

²To minimize the required number of objects in Figure 4, we fix $\tau_x = 4$ and $\tau_y = 2$.

to prune the amount of objects necessary to be tested during the query execution.

5.2 Query Processing and Index Refinement

Having defined QUASII’s data structure, we discuss how it is incrementally built and maintained as a side effect of each query.

Query Processing. Algorithm 1 shows the pseudo-code for query processing. Each query traverses the d -level structure depth-first, starting from the first level (having x -slices). Because the slices are sorted, QUASII performs a binary search (Line 3) to find the starting slice. It then scans all the slices $S[i]$ within the query range on the current dimension (i.e., while the loop conditions in Line 4 hold). The loop conditions guarantee that each slice $S[i]$ intersects q only in the current dimension. To discard potential false positive slices early, Line 5 checks if its actual boundaries ($S[i].box$) also intersect with the query range.

Next, QUASII potentially refines $S[i]$ (Line 6), which may be further sliced into multiple more fine-grained slices S'' if it is larger than the maximum size threshold τ (discussed in the next algorithm). In Lines 7–16, QUASII traverses (potentially refined) slices S'' . For each $s \in S''$, it either checks all s objects for intersection in case of the bottom level or recursively proceeds querying its children based on the next level/dimension (a default child is assigned to a not fully refined slice, Line 15). Finally, all the newly created slices are accumulated in S' (Line 17), appended to S (Line 19), and re-sorted (Line 20). The slices are sorted based on their ids , i.e., the position (index) of the first slice’s object in the data array.

Index Refinement. With each query, QUASII attempts to refine all query intersecting slices (i.e., Line 6 in Algorithm 1). Algorithm 2 provides the simplified pseudo-code for this refinement process. Note that the processing within Algorithm 2 is always based only on the current dimension/level of slice s ($s.l$).

The input slice s is considered for slicing only if it exceeds the threshold τ . Given s is coarse enough, QUASII proceeds with determining the type of slicing based on the intersection between

Algorithm 1: query(query q , data D , slices S , result R)

```

1:  $S' \leftarrow \emptyset$  // to store newly created (refined) slices
2:  $dim \leftarrow S[0].l$  // current level/dimension of slices in  $S$ 
3:  $i \leftarrow \text{binarySearch}(S, \text{lower}(q[dim]))$ 
4: while  $i < |S|$  and  $\text{lower}(S[i].\text{box}[dim]) \leq \text{upper}(q[dim])$ 
   do
5:   if  $q \cap S[i].\text{box} = \emptyset$  then continue
6:    $S'' \leftarrow \text{refine}(S[i], D, q)$  // as per Algorithm 2
7:   for each slice  $s \in S''$  do
8:     if  $q \cap s.\text{box} \neq \emptyset$  then
9:       if  $s.l$  is the bottom level then
10:        for each  $j \in \{s.ids\}$  do
11:          if  $D[j] \cap q \neq \emptyset$  then
12:             $R \leftarrow R \cup D[j]$ 
13:        else
14:          if  $|s.children| = 0$  then
15:            createDefaultChild( $s$ )
16:            query( $q, D, s.children, R$ )
17:           $S' \leftarrow S' \cup S''$ 
18:           $i \leftarrow i + 1$ 
19:  $S \leftarrow S \cup S'$ 
20: sort( $S$ )

```

query q and slice s . It considers three types of slicing. If both q 's lower and upper coordinates are within s , a three-way slicing is performed splitting s into three sub-slices (Line 5). If only one of q 's coordinates is within s , a two-way slicing is performed splitting s into two sub-slices (Line 6). Finally, if q contains s (i.e., both q 's coordinates are outside of s 's bounds), QUASII performs a two-way slicing based on an artificially introduced coordinate.

QUASII iterates through the generated slices and for the ones that still exceed τ (and overlap with the query) it applies additional refinement according to artificially introduced boundaries in Line 10 (it repeats the process recursively until a slice is fully refined in the corresponding dimension). The three- and two-way slicing algorithms (Line 5 and Line 6) reorganize the data (D) following the incremental quick sort strategy introduced in database cracking [18]. In the reorganization process, QUASII also records the information about the boundaries (box) of newly created or modified slices.

Example. Continuing with our example in Figure 4, after refining s_0 into three x sub-slices in Line 6 of Algorithm 1 (and resulting in Figure 4b), QUASII recursively continues with the intersecting (and just refined) slice s_2 based on the y dimension (Figure 4c). As such, s_2 is further refined based on the queried y range and results in three new slices (s_{21} , s_{22} , and s_{23}). In this step, only the objects within the s_2 range ($ids = [1..4]$) are three-way sliced and re-organized in the data array. The two new slices (s_{23} is empty) are appended to the data structure as children of s_2 . They are fully refined (as $|s_{21}| \leq 2$ and $|s_{22}| \leq 2$) and have much smaller MBBs (b_{21} and b_{22} , respectively) than the initial slice cuts. Finally, because it is the bottom level, the objects within s_{22} are checked against the query range and the two qualifying objects (o_4, o_6) are added to the result set (R).

The subsequent query q_2 benefits greatly from previous slicing, as illustrated in Figure 4d. For example, x -slices s_1 and s_2 are skipped completely because query q_2 does not intersect with their MBBs (i.e., test on Line 5 in Algorithm 1). Therefore, QUASII proceeds with the only intersecting slice s_3 , which is not fully refined and requires further slicing. As per Algorithm 2, this time a two-way slicing is performed (at $x = 5.5$) resulting in two finer

Algorithm 2: refine(slice s , data D , query q) \rightarrow slices S

```

1: if  $|s| \leq \tau[s.l]$  then
   return  $\{s\}$ 
2:  $S \leftarrow \emptyset$  // to store refined slices
3:  $t \leftarrow \text{determineSliceType}(s, q)$ 
4: switch ( $t$ )
5:   case both:  $S' \leftarrow \text{sliceThreeWay}(s, q, D)$ 
6:   case one:  $S' \leftarrow \text{sliceTwoWay}(s, q, D)$ 
7:   default:  $S' \leftarrow \text{sliceArtificial}(s, q, D)$ 
8: for each slice  $s \in S'$  do
9:   if  $|s| > \tau[s.l]$  and  $q[s.l] \cap s.\text{box}[s.l] \neq \emptyset$  then
10:     $S'' \leftarrow \text{sliceArtificial}(s, q, D)$ 
11:     $S \leftarrow S \cup S''$ 
12:   else
13:     $S \leftarrow S \cup s$ 
14: return  $S$ 

```

slices (s'_3 and s_4) replacing the previous slice s_3 . Next, QUASII continues with y -based slicing of the fully q_2 -contained slice s'_3 . Since s'_3 reaches the size threshold τ_y , it is not refined further. Finally, the actual data array objects within s'_3 range ($ids = [5..6]$) are checked for intersection with q_2 and the qualifying o_8 is added to the result set.

Artificial Refinement. To produce a balanced hierarchical structure QUASII has to conform with the defined thresholds when forming the slices and using only query boundaries does not meet these requirements. One query is usually not sufficient and we cannot use the subsequent queries for this purpose, as they may interfere with the existing order of the slices. For instance, reorganizing a slice again (that has been organized according to all dimensions) based on the x dimension, may disrupt the previously established partitioning for y and z dimensions.

To address this problem, QUASII reorganizes a slice s (Lines 7 and 10 in Algorithm 2) until it meets a size threshold τ in the corresponding dimension. It achieves this by forcing a two-way slicing based on artificially introduced coordinate and thus splitting the slice into two sub-slices. Given the range (x_l, x_u) , the new coordinate is $c = \lfloor (x_l + x_u)/2 \rfloor$. The two new slices are recursively sliced further until the threshold τ is satisfied.

While more advanced approaches, e.g., based on the concepts from R*-Tree node splitting algorithms [6], would minimize overlap in data structure, they would also significantly increase the cost of incremental strategy. Therefore, QUASII employs the above uniform and low-cost artificial slicing strategy to meet τ thresholds at each of d levels.

Query & Refine. The outcome of QUASII's reorganization strategy are the slices that are within the query range and consequently only the objects in these slices are checked for intersection. However, performing the reorganization following strictly the query's boundaries would produce an incomplete result, as illustrated in Figure 5.

For instance, the object o overlaps with the query range q , however, its lower coordinate is outside the query's boundaries and consequently o would not be identified as a part of the result.

To ensure correct query execution while performing refinement, QUASII employs the query extension technique [40]. More

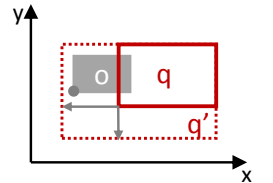


Figure 5: Refinement step: query extension.

precisely, it extends the query for maximum object extent in each dimension, considering lower coordinate. This extension is done only when performing refinement and only within not fully refined slice. Consequently, the query that performs refinement potentially considers more objects for intersection as its range is enlarged. However, this introduces a minimal overhead as the only alternative is the expensive scan of the entire unrefined slice. We apply the same logic for the binary search where, to avoid missing any slices due to the overlap within them, we extend the query range (while performing binary search) for the maximum slice extent in the corresponding dimension.

6 EXPERIMENTAL EVALUATION

In this section, we first describe the experimental setup & methodology and then present a thorough experimental analyses that illustrates the benefits of our incremental approach, both on a real-world neuroscience and synthetic datasets. We start the analyses by outlining the shortcomings of the approaches based on space-oriented partitioning in Section 6.2. We then study the incremental approaches by comparing them with their static counterparts in Section 6.3 and cross-evaluating their performance in Section 6.4. Finally, Section 6.5 describes the sensitivity analyses of QUASII.

6.1 Experimental Setup & Methodology

Hardware. We run our experiments on a Red Hat Enterprise Linux Server release 7.3 machine equipped with 2 Intel Xeon CPU E5-2650L processors at 1.80GHz and 768GB of RAM. Each processor has 12 cores (24 hardware threads) with private L1 (32KB) and L2 (256KB) caches and 30MB of shared L3 cache.

Implementations. All indexing techniques are implemented in C++ and compiled with g++ 4.9.3 with the maximum optimization level. The list below summarizes the implementations that we study:

Scan: performs a full data scan to answer each query.

SFCracker: is our incremental variant of database cracking [18] for spatial data, described in Section 3.1. We use the Z-order as a SFC order. The average farthest distance of neighbours in the Z-order is (slightly) higher than in the Hilbert order [10] (i.e., it has better locality), however, we opt to use the Z-order due to its simplicity and the huge body of work on its efficient range query algorithms [5, 39, 43, 44]. We use 32-bit to represent *zcodes* (i.e., 10 bits per dimension) as a trade-off between memory resources and precision (the number of false positives to be filtered).

SFC: is a static counterpart of SFCracker. In the pre-processing phase, SFC transforms data from multi- to one-dimensional domain and sorts it according to the produced SFCcodes. During querying, a ($3d$) query range is also converted to a $1d$ range and a binary search is used to locate the objects in the $1d$ interval. We employ the same representation of *zcodes* and query optimization as in SFCracker (described in Section 3.1).

QUASII: is our incremental approach discussed in Section 4. We use 60 objects as a node capacity τ_z .

R-Tree: According to our setting, all data is available before querying. Therefore, we use a bulk-loading approach to build the R-Tree index as it reduces overlap and decreases pre-processing time compared to the R-Tree built by inserting one object at a time [26]. For this purpose, we use an efficient STR [26] bulk-loading strategy that balances well the overhead of partitioning the data and query performance. It outperforms Hilbert R-Tree [23] in terms of query performance [26], while its pre-processing cost is not

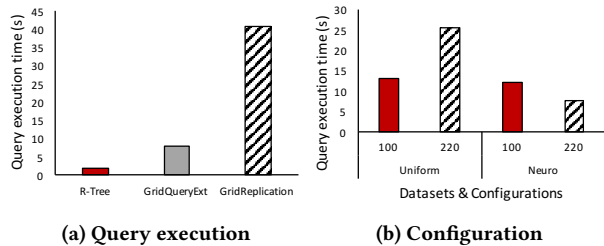


Figure 6: The impact of space-oriented partitioning.

significantly higher [42]. Similarly, TGS [12] and PR-Tree [4] can outperform STR on datasets with extreme skew and aspect ratio, however, they incur considerable overhead for data partitioning. We use the same configuration for node capacity (60) as in QUASII.

Mosaic: corresponds to the space-oriented incremental approach described in Section 3.2.

Grid: is a uniform grid-based index used as a static counterpart of Mosaic. We use query extension [40] technique (as discussed in Section 3.2) to assign an object to a grid cell. We use two configurations with 100 and 220 partitions per dimension for synthetic and neuroscience datasets, respectively. Both configurations are obtained through a parameter sweep.

Dataset and Queries. We use real-world neuroscience and synthetic datasets.

Neuroscience: we use a small part of the rat brain model represented with 450 million cylinders as elements in a volume of $285 \mu m^3$. We approximate the cylinders with MBBs, resulting in the total number of 450 million MBBs with a size of 21GB on disk. Based on the previously described use cases, we synthetically generate queries, each having a fixed volume *qvol* of $10^{-2}\%$ of the queried brain volume and a clustered distribution. We generate 5 query clusters each with 100 queries, where query centers are distributed around the cluster centers following a Gaussian distribution ($\mu = 0, \sigma = qvol$).

Synthetic: we create synthetic datasets by distributing spatial boxes in a space of 10 000 units in each dimension of the $3d$ space. The length of each side of each box is determined uniform randomly between 1 and 10 for 99% of the objects, while 1% of the objects has a side ranging from 10 - 1000 units. The spatial elements are distributed according to a uniform distribution. The datasets have 500 million and 1 billion elements (size on disk 22.5GB and 45GB). For completeness and to test non-skewed cases, we generate *uniform* workload. The uniform workload contains up to 10 000 uniformly distributed queries. To have range queries of different selectivity, we vary *qvol*: $10^{-3}\%$, $10^{-1}\%$, 1%, and 10% of the universe.

6.2 Space-oriented Partitioning Challenges

Both, Mosaic and SFCracker (introduced in Section 3), use space-oriented partitioning at their core — Mosaic partitions space, while SFCracker assigns the SFCcodes using a uniform grid. Before we start the analysis of incremental approaches we experimentally demonstrate the shortcoming of space-oriented partitioning — the overhead introduced with data assignment strategy — since it also affects incremental solutions. Further on, we illustrate why a static approach based on space-oriented partitioning, such as a uniform grid, is not a suitable replacement for an incremental index despite having a comparatively cheap pre-processing step (once properly configured).

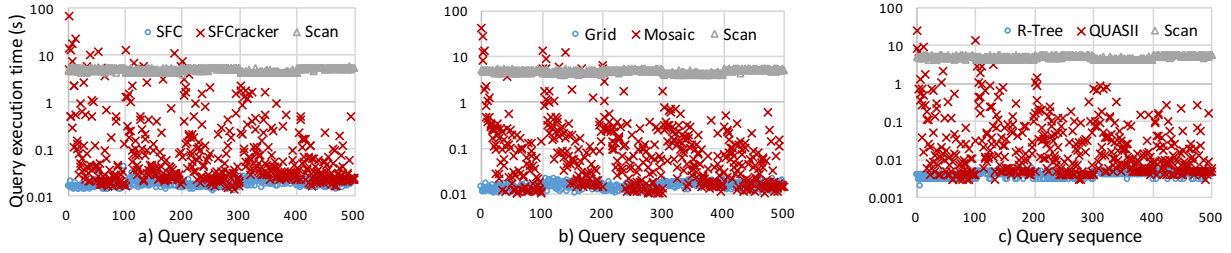


Figure 7: Convergence of a) one-dimensional, b) space-oriented c) data-oriented based approaches.

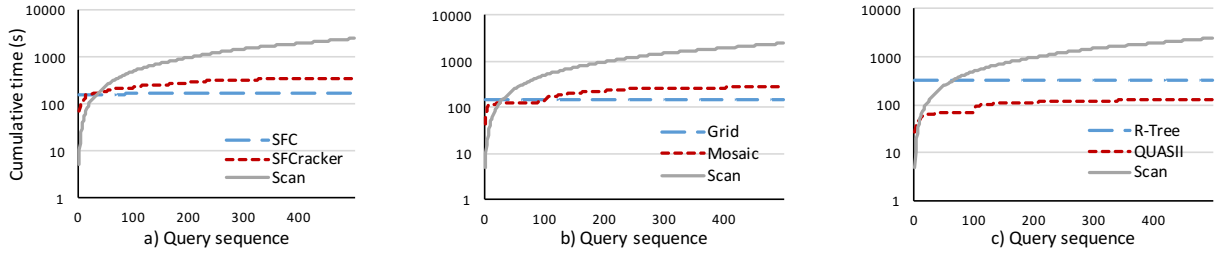


Figure 8: Cumulative time of a) one-dimensional, b) space-oriented c) data-oriented based approaches.

Data Assignment. In the first experiment we illustrate the impact of data assignment strategies by comparing the performance of Grid and R-Tree. We use two variants of the Grid approach: GridQueryExt avoids the objects replication by using the query extension technique – it assigns an object to the grid partition based on its center, while GridReplication replicates the objects – it assigns an object to all the overlapping partitions.

Figure 6a) shows the results of the experiments where we execute 500 clustered queries of selectivity 0.01% on the neuroscience dataset. GridReplication is heavily affected by object replication which increases the number of objects necessary to be checked for intersection and introduces an expensive de-duplication step (needed due to objects replication). GridQueryExt achieves better performance, however, it still considers $3.1\times$ more objects for intersection than the R-Tree as it extends the initial query for the maximum object extent. The R-Tree clearly outperforms both GridReplication and GridQueryExt with a speedup of $19.4\times$ and $3.7\times$ respectively.

Configuration. In the second experiment we demonstrate the difficulty to configure the grid-based approaches. We use two datasets with identical extent and number of elements but different data distributions: Uniform (uniform distribution, synthetic dataset) and Neuro (skewed distribution, the neuroscience dataset). We use the same experimental setup as for the previous experiment. The best configuration (number of partitions per dimension) is 100 for Uniform and 220 for the Neuro dataset and is determined in a parameter sweep. We measure the execution time when using both configurations for each dataset and illustrate the results in Figure 6b).

Although both datasets have the same number of elements and extent, the best configuration significantly depends on the data distribution – the neuroscience dataset requires more partitions compared to the Uniform dataset since it has the very dense regions that require fine grained partitioning. Furthermore, the grid configuration significantly affects performance – the grid performance on the Uniform dataset deteriorates notably when using the Neuro dataset configuration and vice versa.

Summary. Space-oriented partitioning introduces performance penalties. Depending of data assignment strategy, we either consider more elements or suffer from replication. Additionally, the grid configuration is non-trivial and using the wrong one has a detrimental impact on the execution time. In practice we have to use a parameter sweep to find the configuration for a given workload. Consequently, grid configuration turns into a time-consuming process, increasing data-to-insight time.

6.3 Incremental versus Static

We first analyze the incremental approaches by comparing their performance with the performance of their static counterparts (introduced in Section 6.1). Each static approach has similar properties as its incremental counterpart, however, it involves necessary pre-processing. We categorize the approaches according to these properties as a) one-dimensional, b) space-oriented and c) data-oriented approaches. For each category we present the performance of the incremental approach, its static counterpart and Scan. We first evaluate if and when the approaches converge to the performance of their static counterparts and then analyze the overhead of the incremental strategy. For this purpose we execute the clustered query workload with 500 queries of selectivity 0.01% on the *neuroscience* dataset.

Convergence. In the first experiment we evaluate the convergence of the incremental approaches – how fast an approach converges to the execution time of a fully indexed dataset. Figure 7 measures the execution time of each query for all approaches.

The results show five peaks in execution time, one for each query cluster. The execution of the first cluster of queries (and the associated processing of the data) takes the longest as no index structure exists at the beginning. The first queries therefore exceed the cost of Scan, because at this point, the entire dataset has to be scanned along with building partial index structures. Subsequent queries within a cluster use a partial index and thus execute in less time than a full scan, but take longer than queries on the static approach. This process continues as queries in the

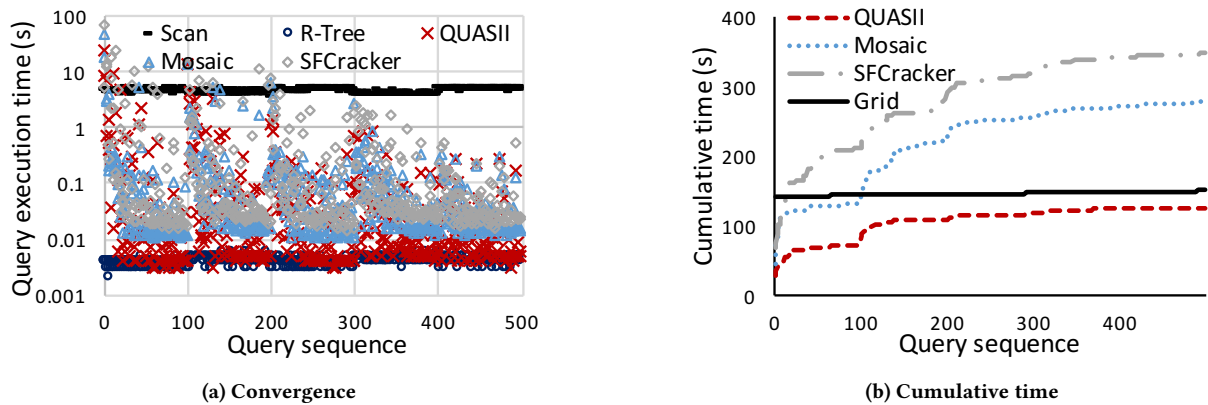


Figure 9: Comparative analysis of incremental approaches.

same cluster further refine the index. Queries in one cluster not only refine the index locally but also carry out limited, global refinement. The queries in a subsequent cluster thus benefit from previous clusters and execute faster. As the index converges to its full structure, the query execution time approaches that of the static approach.

Cumulative Response Time. While in the previous set of experiments we measure the individual query performance, in this analysis we measure the cumulative execution time (including index building step for the static approaches). Figure 8 illustrates the experimental results.

Similar to the convergence experiment, the query clusters are visible: the cumulative response time jumps each time the experiment moves to a new cluster. The most expensive is the transition from the first to the second cluster while subsequent transitions become less evident as the index becomes more refined.

The cumulative cost of SFCracker is comparatively high and, crucially, with a very expensive first query. One reason is that the first query takes 12.9% of the total pre-processing by assigning the objects to the grid cells and calculating the *zcode* values for the entire dataset. Adding to this the cost of cracking, the total execution time of the first query grows to 43% of the total pre-processing time. More precisely, in order to minimize the overhead introduced by the transformation to $1d$ space, we partition the $1d$ query range into sub-intervals that tightly cover its original $3d$ range. This optimization [43] results in a high number of small intervals per query — on average 197. As a consequence, the first queries crack the previously uncracked area into a number of small adjacent intervals and therefore reorganize significant amounts of data.

The static (SFC) index, on the other hand, is not substantially slower for the first queries or, put differently, the building cost of SFC is not much higher than the first query of SFCracker. In fact, the cumulative execution time of SFCracker exceeds the one of SFC after 23 queries already. The incremental approach SFCracker thus does not offer a considerable benefit over SFC.

The incremental strategy of Mosaic is less expensive compared to SFCracker — the objects within the partition queried are potentially reassigned to the eight newly created partitions based on their location. Therefore, it takes Mosaic longer, i.e., 100 queries, before it exceeds the cumulative time of the static Grid. However, its cumulative execution time is still considerable with the biggest overhead being its top-down incremental strategy. The top-down strategy ensures fast convergence but it also introduces overhead

as the data in frequently queried areas is re-partitioned multiple times until Mosaic reaches its final level of refinement.

QUASII, at the same time, does not exceed the cumulative execution of the R-Tree in our experiments. Even after 500 executed queries, the cumulative execution time for QUASII is 39.4% of that of the R-Tree. The main benefit comes from its partial reorganization strategy where the objects are gradually reorganized within the query boundaries, as opposed to the complete sort.

Summary. While all the incremental approaches reach the performance of their static counterparts, the incremental strategies of SFCracker and Mosaic are comparatively expensive. As we show for SFCracker, the major bulk of work has to be done when executing the first query — as the data needs to be transformed to $1d$ space and a single query has to perform multiple cracking operations to avoid performance penalties due to the transformation to $1d$ space. Mosaic increases its cumulative time considerably due to its top-down partitioning strategy — it reorganizes data in frequently queried areas multiple times until it reaches its final level of refinement. Only the cumulative execution time of QUASII does not exceed the one of its static counterpart, the R-Tree, in our experiments.

6.4 Comparative Analysis

We now compare the performance of incremental approaches. We use the same setup as previously and measure the convergence of execution time as well as the cumulative execution time.

Convergence. Figure 9a) depicts the single query execution time for all the incremental approaches compared with the R-Tree and Scan. We use the R-Tree approach as a reference because it is the fastest approach among the static indexes for the workloads tested. We analyze the execution time of the first query and then focus on the performance of the converged data structure.

The execution time of the first query determines data-to-insight time and thus has to be as small as possible. Among the incremental approaches, SFCracker has the most expensive first query due to the transformation of data to the $1d$ space. Mosaic’s first query is faster, but still expensive as it has to reassign all the objects to new partitions, examining all three coordinates. Finally, QUASII has the least expensive first query due to the nested data reorganization — the number of objects necessary to be examined and reorganized becomes smaller as more dimensions are taken into account: all objects are scanned on the x -dimension, but on the y -dimension only the objects with a x -value satisfying the

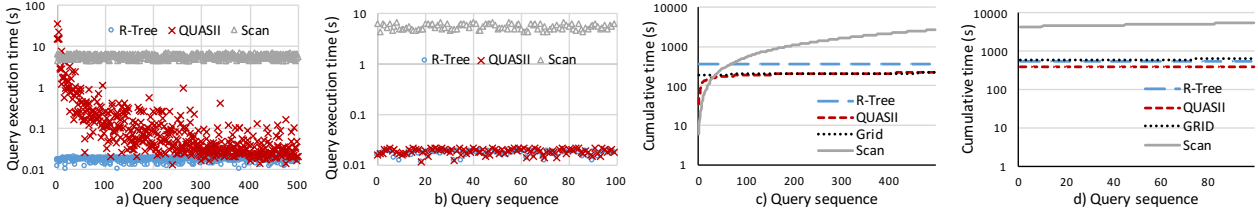


Figure 10: Convergence and cumulative time: the first 500 (a & c) and last 100 (b & d) queries.

query will be scanned (accordingly for the z-dimension). Overall, Scan is 13.7, 9.2 and 4.6 times faster compared to SFCracker, Mosaic and QUASII respectively, when executing the first query.

Among the incremental approaches, only QUASII attains the query execution time of R-Tree on a fully converged index. Mosaic and SFCracker have at their core space-oriented partitioning and therefore, their performance is affected by the data assignment strategy as well as the skew in distribution, as Section 6.2 shows. SFCracker additionally transforms data to 1d domain and thus cannot preserve spatial proximity to the same extent as the other approaches. Consequently, QUASII outperforms Mosaic and SFCracker with a speedup of 3.68x and 4.9x respectively for the average execution time of a query in a fully refined area.

Cumulative Execution Time. We use the cumulative execution time as metric to evaluate the decrease in the data-to-insight time as well as the "break-even" point — the point when the cumulative cost of incremental exceeds that of static indexing — to assess the quality of an incremental index. Figure 9b) shows the experimental results. We use Grid as a reference since it has the smallest cumulative execution time among the static approaches — its pre-processing step is comparatively cheap (once its optimal configuration is determined).

As discussed in Section 6.3, SFCracker and Mosaic have comparatively expensive strategies and thus reach the performance of Grid after 13 and 100 queries respectively. Grid, on the other hand, compared to QUASII, has not amortized its building cost after 500 queries. More precisely, QUASII reaches 84% of the Grid cumulative execution time and, more importantly, it achieves 3.66x faster query performance for completely refined areas. QUASII executes the first query the fastest and consequently achieves the highest decrease in data-to-insight time — 5.1x and 11.4x compared to Grid and R-Tree.

For single query execution, the major benefit of QUASII comes from its data-oriented partitioning. Similar to R-Tree, it adjusts to the distribution of the data and, as opposed to Grid and SFC, it does not replicate the objects or extend the query. It additionally keeps the data in multidimensional space and does consequently not suffer from decrease in dimensionality. Its low cumulative cost is mostly attributed to its incremental strategy. QUASII does not sort all objects, but rather reorganizes them within the specific bounds, gradually curbing the amount of data necessary to be reorganized.

Summary. QUASII outperforms other incremental approaches with respect to the convergence of execution time and cumulative time. It achieves performance comparable to the R-Tree (the fastest static approach) in the areas of the dataset where enough queries have been executed, while not exceeding the cumulative time of Grid (the static approach with the least expensive pre-processing) or the R-Tree. Its major benefits come from the data-oriented partitioning and the nested reorganization strategy which reorganizes precisely the data touched and used.

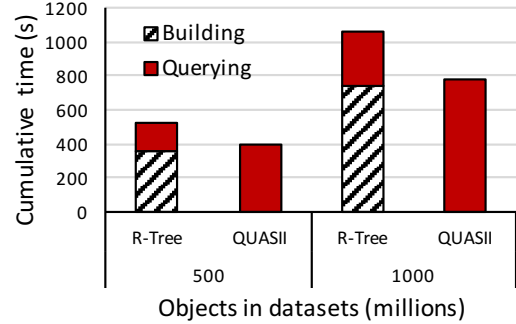


Figure 11: Analysis of QUASII: scalability.

6.5 Analysis of QUASII

In this section we focus on QUASII. We evaluate its performance on the workloads other than neuroscience, analyze its scalability and the impact of query selectivity.

6.6 Uniform Workload

In the previous analyses we used workloads with query clusters that show the benefit of incremental approaches: the index quickly converges to the final performance as the queries are targeting the same areas. In this experiment we evaluate the performance of QUASII for a uniform workload. We execute 10000 uniformly distributed queries of selectivity 0.1% on the dataset with uniform distribution and 500M elements. We compare the performance of QUASII with Scan and R-Tree and additionally consider Grid for the cumulative execution time. Figure 10 illustrates both convergence and cumulative time for the first 500 and last 100 queries of the workload.

None of the first 500 queries is executed on a completely refined index. Starting with the 300th query, however, the single query execution is close to the final performance. Among the last 100 queries, 64 are executed on a completely refined index. The performance of queries on the refined structure is equal or very close to the performance of the R-Tree, i.e., on average 7.5% slower than the R-Tree.

After 10000 executed queries QUASII reaches 75% and 63.8% of the cumulative time of the R-Tree and Grid approaches respectively (*y* axis is in log scale). Likewise, it decreases data-to-insight time by 10.3x and 5.6x compared to R-Tree and Grid. Although the pre-processing step of Grid is significantly cheaper compared to the R-Tree, its cumulative time deteriorates with more queries executed due to the expensive single query performance.

6.7 Performance Trends

In the following experiment we evaluate the scalability of QUASII by executing 10000 queries of selectivity 0.1% on datasets with

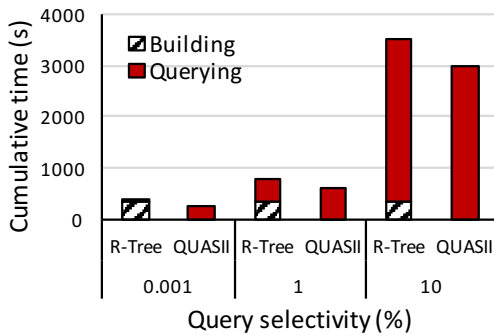


Figure 12: Analysis of QUASII: impact of selectivity.

500 million and 1 billion elements. In Figure 11 we compare the cumulative time of QUASII with R-Tree, where we additionally split the execution time of R-Tree into Building and Querying.

After 10000 executed queries QUASII reaches 75% and 73.7% of the cumulative time of the R-Tree with datasets of 500M and 1B elements respectively. By the time the R-Tree finishes its building process QUASII has executed around 8000 queries in both cases. QUASII also decreases data-to-insight time by 10.3x (on the 500M dataset) and 10.6x (on the 1B dataset) compared to the R-Tree. As illustrated in this experiment, QUASII maintains the performance trends as the dataset size increases.

6.8 Impact of Selectivity

In this set of experiments we evaluate the impact of query selectivity on the performance of QUASII. We measure the cumulative time for a uniform workload: 500M dataset and 5000 queries of 0.001%, 1%, and 10% selectivity. Figure 12 illustrates the results where we consider both the R-Tree and QUASII.

Intuitively, a static index (R-Tree) takes more time to amortize its building cost when executing 0.001% selectivity queries. On the other hand, the lower selectivity queries (10%) touch and reorganize a significant amount of data and QUASII thus reaches the break-even point with the R-Tree faster. Overall, after 5000 executed queries, QUASII reaches 68.8%, 79.8% and 85.6% of the cumulative time of the R-Tree for queries with 0.001%, 1%, and 10% selectivity.

7 RELATED WORK

To the best of our knowledge, no incremental strategy has been proposed to spatial indexing in main memory. While recently an incremental indexing technique has been proposed for exploration of multiple spatial datasets [35], the addressed problem is different (spatial search within multiple datasets) and the proposed solution focuses solely on disk-based efficiency, i.e., reducing the number of expensive disk I/O operations. Nevertheless, there has been considerable interest in incremental data processing within relational databases. Therefore, before giving an overview of related (but not incremental) spatial indexing techniques, we briefly describe incremental approaches used by relational database systems.

7.1 Relational Incremental Indexing

Incremental (one-dimensional) indexing techniques are extensively studied in database cracking [16, 18, 19] and adaptive merging [13, 14]. The former partially sorts keys in an in-memory

relation, essentially performing quicksort. The latter, adaptive merging, takes the idea further and devises an incremental, external sort to make use of external memory as well.

Driven by the same goal (minimize data-to-insight time), novel systems have been proposed that bypass data pre-processing step and execute queries on raw data files. Instead, auxiliary data structures are built incrementally so that the most popular data subsets are serviced at the speeds of fully loaded/indexed data. For example, NoDB [2], RAW [25], and ViDa [24] incrementally build positional maps to track the position of frequently accessed data fields. This enables the systems to “jump” to previously queried data regions and potentially reduce the costs of tokenizing and parsing raw data sources.

7.2 Spatial Indexing

Research in indexing spatial data has produced numerous approaches in past decades [11]. In the following we briefly review spatial indexing approaches that we group into three classes depending on how amenable they are to incremental indexing.

One-dimensional Indexing. One way to address the curse of dimensionality in the context of spatial data is to transform it from multi- to one-dimensional domain. Typical methods to perform this transformation are space-filling curves like the Z-order [34], the Hilbert curve [21], and the Gray-code curve [9]. They impose a total order and preserve spatial proximity between objects - if the objects are close in higher-dimensional space, they are also close on the space-filling curve - reasonably well. Given such a mapping of spatial data, the existing 1d access methods, such as B-Tree [5], can be used for querying.

Data-oriented Indexing. The data-oriented partitioning approaches create an index structure taking into consideration the data distribution, where the prominent representatives are the KD-Tree [7], the R-Tree [15], and its variants. The R-Tree, arguably the most important data-oriented spatial index, is multi-dimensional generalization of the B-Tree which recursively encloses objects in minimum bounding rectangles (MBRs). The basic R-Tree definition faces the problems of overlap and dead space, both detrimental to query execution performance [15, 42]. Multiple approaches have been devised to address the issue. The R*-Tree [6], for example, uses an improved version of the node split algorithm and reinsertion of objects while the R+-Tree [37] tries to avoid overlap through the duplication of MBRs (leading to a bigger index). A priori knowledge of the entire dataset may help to reduce the above problems of the R-Tree by packing spatially close objects on the same disk page. The Hilbert R-Tree [23] achieves this using the Hilbert curve, Sort-Tile-Recursive (STR) [26] recursively sorts objects in all dimensions to do so, while Top-down Greedy Split (TGS) [12] recursively splits the data set into partitions minimizing the area on each level.

Adaptive index structures [41] rearrange the nodes of data-oriented hierarchical indexes (including the R-Tree index) in response to queries so that they can be accessed sequentially on disk. However, this reorganization is performed to improve query performance by optimizing disk-access without taking into consideration data-to-insight time.

A recently proposed partitioning mechanism for large-scale spatial data also adapts to an incoming query workload [3]. In contrast to QUASII, however, its primary goal is not to minimize data-to-insight time (as all necessary data structures are still built during pre-processing), but to efficiently accommodate changes in data and workload. Also, it considers a distributed setting.

Space-oriented Indexing. The space-oriented partitioning approaches assign the data to the partitions based on space, regardless of data distribution. A typical representative is the uniform grid that partitions the space uniformly into partitions of equal size [38]. Similarly, the Quadtree [36] and its variant for 3d space, the Octree [20], recursively divide space into four/eight partitions of equal size to build a hierarchy of partitions. Further approaches, for example the grid file [32], use a non-uniform grid to better accommodate skew in the data (and to also optimize for disk access). The downside, is a more complex query execution due to the cells of different size and location. The two level grid file [17] addresses the issue by introducing an additional level with a coarser grid. Still, the overhead of testing the query against multiple cells can be substantial.

8 CONCLUSIONS

The advances in data acquisition technologies and supercomputing for large-scale simulations rapidly increase the amounts of spatial data generated and collected. This data helps the scientist tremendously to gain insights and understand natural phenomena, however, at the same time, it leaves them with the challenge of analyzing it. Known approaches to spatial indexing have two major drawbacks with respect to exploratory analyses. First, they require a time-consuming pre-processing step that delays analyses. Second, given the massive amounts of data, a scientist frequently only analyzes a small fraction of it and consequently indexing the entirety of the data does not always pay off.

In this paper we propose a novel incremental index for the exploration of spatial data, where the ultimate goal is to let the scientists perform exploratory analyses as soon as the data is available, while using their queries to incrementally index the data. Our approach, QUASII, reduces data-to-insight time and curbs the cost of incremental indexing, by gradually and partially sorting the data, while producing a data-oriented hierarchical structure. As our experiments show, QUASII reduces the data-to-insight time by up to a factor of 11.4x, while its performance converges to that of the fastest state-of-the-art static indexes.

ACKNOWLEDGEMENTS

We would like to thank the reviewers, the DIAS laboratory members, and Georgios Chatzopoulos for their comments and suggestions on how to improve the paper. This work is partially funded by the EU FP7 programme (ERC-2013-CoG), Grant No 617508 (ViDa) and EU Horizon 2020, GA No 720270 (HBP SGA1).

REFERENCES

- [1] C.L. Abad, N. Roberts, Yi Lu, and R.H. Campbell. 2012. A storage-centric analysis of MapReduce workloads: File popularity, temporal locality and arrival patterns. In *IISWC*. 100–109.
- [2] Ioannis Alagiannis, Renata Borovica, Miguel Branco, Stratos Idreos, and Anastasia Ailamaki. 2012. NoDB: efficient query execution on raw data files. In *SIGMOD*. 241–252.
- [3] Ahmed M. Aly, Ahmed R. Mahmood, Mohamed S. Hassan, Walid G. Aref, Mourad Ouzzani, Hazem Elmeleegy, and Thamer Qadah. 2015. AQWA: Adaptive Query-Workload-Aware Partitioning of Big Spatial Data. *PVLDB* 8, 13 (2015), 2062–2073.
- [4] Lars Arge, Mark de Berg, Herman J. Haverkort, and Ke Yi. The Priority R-tree: a practically efficient and worst-case optimal R-tree. In *SIGMOD '04*.
- [5] Rudolf Bayer. 1997. The Universal B-Tree for Multidimensional Indexing: General Concepts. In *WWCA*. 198–209.
- [6] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R⁺-tree: An Efficient and Robust Access Method for Points and Rectangles. In *SIGMOD*.
- [7] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *CACM* 18, 9 (1975), 509–517.
- [8] Yanpei Chen, Sara Alspaugh, and Randy Katz. 2012. Interactive analytical processing in big data systems: A cross-industry study of MapReduce workloads. *PVLDB* 5, 12 (2012), 1802–1813.
- [9] Christos Faloutsos. 1988. Gray codes for partial match and range queries. *TSE* 14, 10 (1988), 1381–1393.
- [10] Christos Faloutsos and Shari Roseman. 1989. Fractals for secondary key retrieval. In *PODS*. 247–252.
- [11] Volker Gaede and Oliver Guenther. 1998. Multidimensional Access Methods. *CSUR* 30, 2 (1998).
- [12] Yván J. García, Mario A. López, and Scott T. Leutenegger. 1996. A Greedy Algorithm for Bulk Loading R-trees. In *GIS*.
- [13] G. Graefe and H. Kuno. 2010. Adaptive Indexing for Relational Keys. In *ICDEW*.
- [14] Goetz Graefe and Harumi Kuno. 2010. Self-selecting, Self-tuning, Incrementally Optimized Indexes. In *EDBT*.
- [15] Antonin Guttmann. 1984. R-trees: a dynamic index structure for spatial searching. In *SIGMOD*. 47–57.
- [16] Felix Halim, Stratos Idreos, Panagiotis Karras, and Roland H. C. Yap. 2012. Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores. In *VLDB*.
- [17] Klaus Hinrichs. 1985. Implementation of the Grid File: Design Concepts and Experience. *BIT* 25, 4 (1985).
- [18] Stratos Idreos, Martin L Kersten, and Stefan Manegold. Database Cracking.. In *CIDR*.
- [19] Stratos Idreos, Stefan Manegold, Harumi Kuno, and Graefe Goetz. 2011. Merging what's cracked, cracking what's merged: adaptive indexing in main-memory column-stores. In *VLDB*.
- [20] Chris L Jackins and Steven L Tanimoto. 1980. Oct-trees and their use in representing three-dimensional objects. *Comp. Graphics and Image Proc.* 14, 3 (1980), 249–270.
- [21] Hosagrahar V Jagadish. 1990. Linear clustering of objects with multiple attributes. *SIGMOD Rec.* 19, 2 (1990), 332–342.
- [22] Christian S Jensen, Dan Lin, and Beng Chin Ooi. 2004. Query and update efficient B⁺-tree based indexing of moving objects. In *VLDB*. 768–779.
- [23] Ibrahim Kamel and Christos Faloutsos. 1993. Hilbert R-tree: An improved R-tree using fractals. In *VLDB*. 500–509.
- [24] Manos Karpathiotakis, Ioannis Alagiannis, Thomas Heinis, Miguel Branco, and Anastasia Ailamaki. Just-in-time data virtualization: Lightweight data management with ViDa. In *CIDR'15*.
- [25] Manos Karpathiotakis, Miguel Branco, Ioannis Alagiannis, and Anastasia Ailamaki. 2014. Adaptive query processing on RAW data. *PVLDB* 7, 12 (2014), 1119–1130.
- [26] Scott T Leutenegger, Mario Lopez, Jeffrey Edgington, et al. 1997. STR: A simple and efficient algorithm for R-tree packing. In *ICDE*. 497–506.
- [27] Henry Markram et al. 2011. Introducing the Human Brain Project. *Procedia Computer Science* 7, 1. FET '11.
- [28] Henry Markram et al. 2015. Reconstruction and simulation of neocortical microcircuitry. *Cell* 163, 2 (2015), 456–492.
- [29] Bongki Moon, Hosagrahar V Jagadish, Christos Faloutsos, and Joel H Saltz. 2001. Analysis of the clustering properties of the Hilbert space-filling curve. *TKDE* 13, 1 (2001), 124–141.
- [30] EarthData NASA. <https://earthdata.nasa.gov/>.
- [31] Actueel Hoogte Bestand Nederland. 2017. *AHN datasets*. <http://www.ahn.nl>.
- [32] J. Nievergelt, Hans Hinterberger, and Kenneth C. Sevcik. 1984. The Grid File: An Adaptable, Symmetric Multikey File Structure. *TODS* 9, 1 (1984).
- [33] OpenStreetMap. <https://www.openstreetmap.org>.
- [34] Jack A Orenstein and Tim H Merrett. 1984. A class of data structures for associative searching. In *PODS*. 181–190.
- [35] Mirjana Pavlovic, Eleni Tzirita Zacharotou, Darius Sidlauskas, Thomas Heinis, and Anastasia Ailamaki. 2016. Space Odyssey: Efficient Exploration of Scientific Data. In *ExploreDB*. 12–18.
- [36] Hanan Samet. 1984. The quadtree and related hierarchical data structures. *CSUR* 16, 2 (1984), 187–260.
- [37] Timos Sellis, Nick Roussopoulos, and Christos Faloutsos. 1987. The R⁺-tree: A dynamic index for multi-dimensional objects. In *VLDB*.
- [38] Darius Sidlauskas, Simonas Šaltenis, et al. 2009. Trees or grids? Indexing moving objects in main memory. In *SIGSPATIAL*. 236–245.
- [39] Tomáš Skopal, Michal Krátký, Jaroslav Pokorný, and Václav Sňámel. 2006. A new range query algorithm for Universal B-trees. *Information Systems* 31, 6 (2006), 489–511.
- [40] Emmanuel Stefanakis, Yannis Theodoridis, Timos Sellis, and Yuk-Cheung Lee. 1997. Point representation of spatial objects and query window extension: A new technique for spatial access methods. *GIScience* 11, 6 (1997), 529–554.
- [41] Yufei Tao and Dimitris Papadias. 2002. Adaptive Index Structures. In *VLDB*.
- [42] Farhan Tauheed, Laurynas Biveinis, Thomas Heinis, Felix Schürmann, Henry Markram, and Anastasia Ailamaki. 2012. Accelerating Range Queries For Brain Simulations. In *ICDE*. 941–952.
- [43] Hermann Tropf and H. Herzog. 1981. Multidimensional Range Search in Dynamically Balanced Trees. *Angewandte Informatik* 23, 2 (1981), 71–77.
- [44] Peter van Oosterom, Oscar Martinez-Rubi, Milena Ivanova, Mike Hörhammer, Daniel Geringer, Siva Ravada, Theo Tijssen, Martin Kodde, and Romulo Goncalves. 2015. Massive point cloud data management: Design, implementation and execution of a point cloud benchmark. *Computers & Graphics* 49 (2015), 92–125.
- [45] Kostas Zoumpatianos, Stratos Idreos, and Themis Palpanas. 2014. Indexing for interactive exploration of big data series. In *SIGMOD*. 1555–1566.

Loom: Query-aware Partitioning of Online Graphs

Hugo Firth
School of Computing Science
h.firth@ncl.ac.uk

Paolo Missier
School of Computing Science
paolo.missier@ncl.ac.uk

Jack Aiston
School of Computing Science
j.aiston@ncl.ac.uk

ABSTRACT

As with general graph processing systems, partitioning data over a cluster of machines improves the scalability of graph database management systems. However, these systems will incur additional network cost during the execution of a query workload, due to inter-partition traversals. Workload-agnostic partitioning algorithms typically minimise the likelihood of any edge crossing partition boundaries. However, these partitioners are sub-optimal with respect to many workloads, especially queries, which may require more frequent traversal of specific subsets of inter-partition edges. Furthermore, they are largely unsuited to operating incrementally on dynamic, growing graphs.

We present a new graph partitioning algorithm, Loom, that operates on a stream of graph updates and continuously allocates the new vertices and edges to partitions, taking into account a query workload of graph pattern expressions along with their relative frequencies. First we capture the most common patterns of edge traversals which occur when executing queries. We then compare sub-graphs, which present themselves incrementally in the graph update stream, against these common patterns. Finally we attempt to allocate each match to single partitions, reducing the number of inter-partition edges within frequently traversed sub-graphs and improving average query performance.

Loom is extensively evaluated over several large test graphs with realistic query workloads and various orderings of the graph updates. We demonstrate that, given a workload, our prototype produces partitionings of significantly better quality than existing streaming graph partitioning algorithms Fennel & LDG.

1 INTRODUCTION

Subgraph pattern matching is a class of operation fundamental to many “real-time” applications of graph data. For example, in social networks [9], and network security [3]. Answering a subgraph pattern matching query usually involves exploring the subgraphs of a large, labelled graph G then finding those which match a small labelled graph q . Fig.1 shows an example graph G and a set of query graphs Q which we will refer to throughout. **Efficiently partitioning large, growing graphs to optimise for given workloads of such queries** is the primary contribution of this work.

In specialised graph database management systems (GDBMS), pattern matching queries are highly efficient. They usually correspond to some index lookup and subsequent traversal of a small number of graph edges, where edge traversal is analogous to pointer dereferencing. However, as graphs like social networks may be both large and continually growing, eventually they saturate the memory of a single commodity machine and must be partitioned and distributed. In such a distributed setting, queries which require inter-partition traversals, such as q_2 in Fig. 1, incur network communication costs and will perform poorly. A

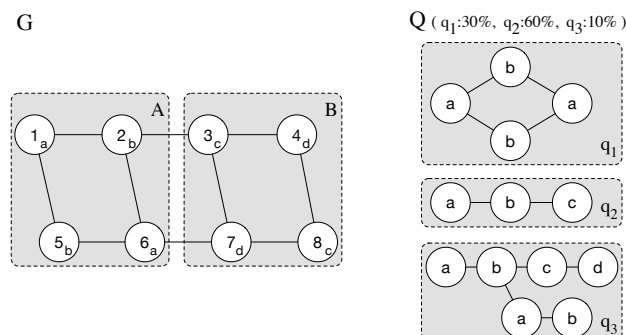


Figure 1: Example graph G with query workload Q

widely recognised approach to addressing these scalability issues in graph data management is to use one of several k -way balanced graph partitioners [2, 11, 14, 18, 30–32]. These systems distribute vertices, edges and queries evenly across several machines, seeking to optimise some global goal, e.g. minimising the number of edges which connect vertices in different partitions (a.k.a *min. edge-cut*). In so doing, they improve the performance of a broad range of possible analyses.

Whilst graphs partitioned for such global measures mostly work well for global forms of graph analysis (e.g. Pagerank), no one measure is optimal for all types of operation [28]. In particular, the workloads of pattern matching query workloads, common to GDBMS, are a poor match for these kinds of partitioned graphs, which we call *workload agnostic*. This is because, intuitively, a min. edge-cut partitioning is equivalent to assuming uniform, or at least constant, likelihood of traversal for each edge throughout query processing. This assumption is unrealistic as a query workload may traverse a limited subset of edges and edge types, which is specific to its graph patterns and subject to change.

To appreciate the importance of a workload-sensitive partitioning, consider the graph of Fig.1. The partitioning $\{A, B\}$ is optimal for the balanced min. edge-cut goal, but may not be optimal for the query graphs in Q . For example, the query graph q_2 matches the subgraphs $\{(1, 2), (2, 3)\}$ and $\{(6, 2), (2, 3)\}$ in G . Given a workload which consisted entirely of q_2 queries, every one would require a potentially expensive inter-partition traversal (*ipt*). It is easy to see that the alternative partitioning $A' = \{1, 2, 3, 6\}$, $B' = \{4, 5, 7, 8\}$ offers an improvement (*0 ipt*) given such a workload, whilst being strictly worse *w.r.t* min. edge-cut.

Mature research of workload-sensitive online database partitioning is largely confined to relational DBMS [4, 23, 26]

1.1 Contributions

Given the above motivation, we present Loom: a partitioner for online, dynamic graphs which optimises vertex placement to improve the performance of a given stream of sub-graph pattern matching queries.

The simple goals of Loom are threefold: a) to discover patterns of edge traversals which are common when answering queries

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

from our given workload Q ; b) to efficiently detect instances of these patterns in the ongoing stream of graph updates which constitutes an online graph; and c) to assign these pattern matches wholly within an individual partition or across as few partitions as possible, thereby reducing the number of *ipt* and increasing the average performance of any $q \in Q$.

This work extends an earlier “vision” work [7] by the authors, providing the following additional contributions:

- A compact ¹ trie based encoding of the most frequent traversal patterns over edges in G . We show how it may be constructed and updated given an evolving workload Q .
- A method of sub-graph isomorphism checking, extending a recent probabilistic technique[29]. We show how this measure may be efficiently computed and demonstrate both the low probability of false positives and the impossibility of false negatives.
- A method for efficiently computing matches for our frequent traversal patterns in a graph stream, using our trie encoding and isomorphism method, and then assign these matching sub-graphs to graph partitions, using heuristics to preserve balance. Resulting partitions do not rely upon replication and are therefore agnostic to the complex replication schemes often used in production systems.

As online graphs are equivalent to graph streams, we present an extensive evaluation comparing Loom to popular streaming graph partitioners Fennel [31] and LDG[30]. We partition real and synthetic graph streams of various sizes and with three distinct stream orderings: breadth-first, depth-first and random order. Subsequently, we execute query workloads over each graph, counting the number of expensive *ipt* which occur. Our results indicate that Loom achieves a significant improvement over both systems, with between 15 and 40% fewer *ipt* when executing a given workload.

1.2 Related work

Partitioning graphs into k balanced subgraphs is clearly of practical importance to any application with large amounts of graph structured data. As a result, despite the fact that the problem is known to be NP-Hard [1], many different solutions have been proposed [2, 11, 14, 18, 30–32]. We classify these partitioning approaches into one of three potentially overlapping categories: streaming, non-streaming and workload sensitive. Loom is both a streaming and workload-sensitive partitioner.

Non-streaming graph partitioners [2, 14, 18] typically seek to optimise an objective function global to the graph, e.g. minimising the number of edges which connect vertices in different partitions (*min. edge-cut*).

A common class of these techniques is known as multi-level partitioners [2, 14]. These partitioners work by computing a succession of recursively compressed graphs, tracking exactly how the graph was compressed at each step, then trivially partitioning the smallest graph with some existing technique. Using the knowledge of how each compressed graph was produced from the previous one, this initial partitioning is then “projected” back onto the original graph, using a local refinement technique (such as Kernighan-Lin [15]) to improve the partitioning after each step. A well known example of a multilevel partitioner is METIS [14], which is able to produce high quality partitionings for small and

medium graphs, but performance suffers significantly in the presence of large graphs [31]. Other multilevel techniques [2] share broadly similar properties and performance, though they differ in the method used to compress the graph being partitioned.

Other types of non-streaming partitioner include Sheep [18]: a graph partitioner which creates an elimination tree from a distributed graph using a map-reduce procedure, then partitions the tree and subsequently translates it into a partitioning of the original graph. Sheep optimises for another global objective function: minimising the number of different partitions in which a given vertex v has neighbours (*min. communication volume*).

These non-streaming graph partitioners suffer from two main drawbacks. Firstly, due to their computational complexity and high memory usage[30], they are only suitable as offline operations, typically performed ahead of analytical workloads. Even those partitioners which are distributed to improve scalability, such as Sheep or the parallel implementation of METIS (ParMETIS) [14], make strong assumptions about the availability of global graph information. As a result they may require periodic re-execution, i.e. given a dynamic graph following a series of graph updates, which is impractical online [13]. Secondly, as mentioned, partitioners which optimise for such global measures assume uniform and constant usage of a graph, causing them to “leave performance on the table” for many workloads.

Streaming graph partitioners [11, 30, 31] have been proposed to address some of the problems with partitioners outlined above. Firstly, the strict streaming model considers each element of a graph stream as it arrives, efficiently assigning it to a partition. Additionally, streaming partitioners do not perform any refinement, i.e. later reassigning graph elements to other partitions, nor do they perform any sort of global introspection, such as spectral analysis. As a result, the memory usage of streaming partitioners is both low and independent of the size of the graph being partitioned, allowing streaming partitioners to scale to very large graphs (e.g. billions of edges). Secondly, streaming partitioners may trivially be applied to continuously growing graphs, where each new edge or update is an element in the stream.

Streaming partitioners, such as Fennel [31] and LDG [30], make partition assignment decisions on the basis of inexpensive heuristics which consider the local neighbourhood of each new element at the time it arrives. For instance, LDG assigns vertices to the partitions where they have the most neighbours, but penalises that number of neighbours for each partition by how full it is, maintaining balance. By using the local neighbourhood of a graph element e **at the time e is added**, such heuristics render themselves sensitive to the ordering of a graph stream. For example, a graph which is streamed in the order of a *breadth-first traversal* of its edges will produce a better quality partitioning than a graph which is streamed in random order, which has been shown to be pseudo adversarial[31].

In general, streaming algorithms produce partitionings of lower quality than their non-streaming counterparts but with much improved performance. However, some systems, such as the graph partitioner Leopard [11], attempt to strike a balance between the two. Leopard relies upon a streaming algorithm (Fennel) for the initial placement of vertices but drops the “one-pass” requirement and repeatedly considers vertices for reassignment; improving quality over time for dynamic graphs, but at the cost of some scalability. Note that these Streaming partitioners, like their non-streaming counterparts, are workload agnostic and so share those disadvantages.

¹Grows with the size of query graph patterns, which are typically small

Workload sensitive partitioners [4, 23, 24, 26, 28, 32] attempt to optimise the placement of data to suit a particular workload. Such systems may be streaming or non-streaming, but are discussed separately here because they pertain most closely to the work we do with Loom.

Some partitioners, such as LogGP [32] and CatchW [28], are focused on improving graph analytical workloads designed for the *bulk synchronous parallel* (BSP) model of computation². In the BSP model a graph processing job is performed in a number of supersteps, synchronised between partitions. CatchW examines several common categories of graph analytical workload and proposes techniques for predicting the set of edges likely to be traversed in the next superstep, given the category of workload and edges traversed in the previous one. CatchW then moves a small number of these predicted edges between supersteps, minimising inter-partition communication. LogGP uses a similar log of activity from previous supersteps to construct a hypergraph where vertices which are frequently accessed together are connected. LogGP then partitions this hypergraph to suggest placement of vertices, reducing the analytical job’s execution time in future.

In the domain of **RDF stores**, Peng et al. [24] use frequent sub-graph mining ahead of time to select a set of patterns common to a provided SPARQL query workload. They then propose partitioning strategies which ensure that any data matching one of these frequent patterns is allocated wholly within a single partition, thus reducing average query response time at the cost of having to replicate (potentially many) sub-graphs which form part of multiple frequent patterns. Harbi et al. [10] also detect patterns common to workloads of SPARQL queries; their system, called AdPart, redistributes data between partitions over time such that more queries may be executed without *ipt*. However, like Peng et al’s work, AdPart makes significant use of replication. Additionally, AdPart’s re-partitioning approach relies upon an initial input generated by a naive Hash partitioner. Thus, it could potentially be used in conjunction with a streaming system like Loom: a workload-aware initial partitioning reducing the amount of data redistribution required later.

For **RDBMS**, systems such as Schism [4] and SWORD [26] capture query workload samples ahead of time, modelling them as hypergraphs where edges correspond to sets of records which are involved in the same transaction. These graphs are then partitioned using existing non-streaming techniques (METIS) to achieve a min. edge-cut. When mapped back to the original database, this partitioning represents an arrangement of records which causes a minimal number of transactions in the captured workload to be distributed. Other systems, such as Horticulture [23], rely upon a function to estimate the cost of executing a sample workload over a database and subsequently explore a large space of possible candidate partitionings. In addition to a high upfront cost [4, 23], these techniques focus on the relational data model, and so make simplifying assumptions, such as ignoring queries which traverse > 1-2 edges [26] (i.e. which perform nested joins). Larger traversals are common to sub-graph pattern matching queries, therefore its unclear how these techniques would perform given such a workload.

Overall, the works reviewed above either focus on different types of workload than we do with Loom (namely offline analytical or relational queries), or they make extensive use of

replication. Loom does not use any form of replication, both to avoid potentially significant storage overheads [25] and to remain interoperable with the sophisticated replication schemes used in production systems.

1.3 Definitions

Here we review and define important concepts used throughout the rest of the paper.

A **labelled graph** $G = (V, E, L_V, f_l)$ is of the form: a set of vertices $V = \{v_1, v_2, \dots, v_n\}$, a set of pairwise relationships called edges $e = (v_i, v_j) \in E$ and a set of vertex labels L_V . The function $f_l : V \rightarrow L_V$ is a surjective mapping of vertices to labels. Note that throughout this work, for simplicity, we consider only *undirected* graphs. However, all techniques subsequently presented may be extended to directed graphs, which we highlight inline. We view an **online graph** simply as a (possibly infinite) sequence of edges which are being added to a graph G , over time. We consider *fixed width* sliding windows over such a graph, i.e. a sliding window of time t is equivalent to the t most recently added edges. Note that an online may be viewed as a **graph stream** and we use the two terms interchangeably.

A **pattern matching query** is defined in terms of sub-graph isomorphism. Given a pattern graph $q = (V_q, E_q)$, a query should return R : a set of sub-graphs of G . For each returned sub-graph $R_i = (V_{R_i}, E_{R_i})$ there should exist a bijective function f such that: *a*) for every vertex $v \in V_{R_i}$, there exists a corresponding vertex $f(v) \in V_q$; *b*) for every edge $(v_1, v_2) \in E_{R_i}$, there exists a corresponding edge $(f(v_1), f(v_2)) \in E_q$; and *c*) for every vertex $v \in R_i$, the labels match those of the corresponding vertices in q , $l(v) = l(f(v))$. A query workload is simply a multiset of these queries $Q = \{(q_1, n_1) \dots (q_h, n_h)\}$, where n_i is the relative frequency of q_i in Q .

A **query motif** is a graph which occurs, with a frequency of more than some user defined threshold \mathcal{T} , as a sub-graph of query graphs from a workload Q .

A vertex centric **graph partitioning** is defined as a disjoint family of sets of vertices $P_k(G) = \{V_1, V_2, \dots, V_k\}$. Each set V_i , together with its edges E_i (where $e_i \in E_i$, $e_i = (v_i, v_j)$, and $\{v_i, v_j\} \subseteq V_i$), is referred to as a *partition* S_i . A partition forms a proper sub-graph of G such that $S_i = (V_i, E_i)$, $V_i \subseteq V$ and $E_i \subseteq E$. We define the quality of a graph partitioning relative to a given workload Q . Specifically, the number of inter-partition traversals (*ipt*) which occur while executing Q over $P_k(G)$. Whilst min. edge-cut is the standard scale free measure of partition quality [14], it is intuitively a proxy for *ipt* and, as we have argued (Sec. 1), not always an effective one.

1.4 Overview

Once again, Loom continuously partitions an online graph G into k parts, optimising for a given workload Q . The resulting partitioning $P_k(G, Q)$ reduces the *probability* of expensive *ipt*, when executing a random $q \in Q$, using the following techniques.

Firstly, we employ a trie-like datastructure to index all of the possible sub-graphs of query graphs $q \in Q$, then identify those sub-graphs which are motifs, i.e. occur most frequently (Sec. 2). Secondly, we buffer a sliding window over G , then use an efficient graph stream pattern matching procedure to check whether each new edge added to G creates a sub-graph which matches one of our motifs (Sec. 3). Finally, we employ a combination of novel and existing partitioning heuristics to assign each motif matching

²e.g. Pagerank executed using the Apache Giraph framework: <http://bit.ly/2eNVCnv>.

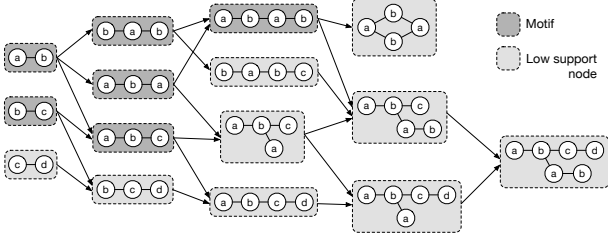


Figure 2: TPSTry++ for Q in fig. 1

sub-graph which leaves the sliding window entirely within an individual partition, thereby reducing ipt for Q (Sec. 4).

2 IDENTIFYING MOTIFS

We now describe the first of the three steps mentioned above, namely the encoding of all query graphs found in our pattern matching query workload Q . For this, we use a trie-like data-structure which we have called the *Traversal Pattern Summary Trie* (TPSTry++). In a TPSTry++, every node represents a graph, while every parent node represents a sub-graph which is common to the graphs represented by its children. As an illustration, the complete TPSTry++ for the workload Q in Fig. 1 is shown in Fig. 2.

This structure not only encodes all sub-graphs found in each $q \in Q$, it also associates a support value p with each of its nodes, to keep track of the relative frequency of occurrences of each sub-graph in our query graphs.

Given a threshold \mathcal{T} for the frequency of occurrences, a *motif* is a sub-graph that occurs at least \mathcal{T} times in Q . As an example, for $\mathcal{T} = 40\%$, Q 's motifs are the shaded nodes in Fig. 2.

Intuitively, a sub-graph of G which is frequently traversed by a query workload should be assigned to a single partition. We can identify these sub-graphs as they form within the stream of graph updates, by matching them against the motifs in the TPSTry++. Details of the motif matching process are provided in Sec.3. In the rest of this section we explain how a TPSTry++ is constructed, given a workload Q .

A TPSTry++ extends a simpler structure, called TPSTry, which we have recently proposed in a similar setting [8]. It employs *frequent sub-graph mining*[12] to compactly encode general labelled graphs. The resulting structure is a Directed Acyclic Graph (DAG), to reflect the multiple ways in which a particular query pattern may extend shorter patterns. For example in Fig. 2 the graph in node $a-b-a-b$ can be produced in two ways, by adding a single $a-b$ edge to either of the sub-graphs $b-a-b$, and $a-b-a$. In contrast, a TPSTry is a tree that encodes the space of possible traversal **paths** through a graph as a conventional trie of strings, where a path is a string of vertex labels, and possible paths are described by a stream of regular path queries [20].

Note that the trie is a relatively compact structure, as it grows with $|L_V|^t$, where t is the number of edges in the largest query graph in Q and L_V is typically small. Also note that the TPSTry++ is similar to, though more general than, Ribiero et al's G-Trie [27] and Choudhury et al's SJ-Tree [3], which use trees (not DAGs) to encode unlabelled graphs and labelled paths respectively.

2.1 Sub-graph signatures

We build the trie for Q by progressively building and merging smaller tries for each $q \in Q$, as shown in Fig. 3. This process relies

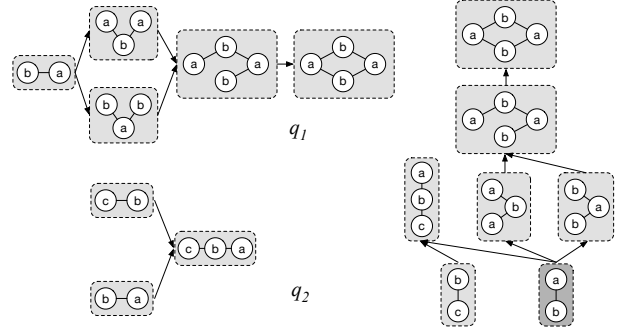


Figure 3: Combining tries for query graphs q_1, q_2

on detecting graph isomorphisms, as any two trie nodes from different queries that contain identical graphs should be merged. Failing to detect isomorphism would result, for instance, in two separate trie nodes being created for the simple graphs $a-b-c$ and $c-b-a$, rather than a single node with a support of 2, as intended. One way of detecting isomorphism, often employed in frequent sub-graph mining, involves computing the lexicographical canonical form for each graph [19], whereby two graphs are isomorphic if and only if they have the same canonical representation.

Computing a graph's canonical form provides strong guarantees, but can be expensive[27]. Instead, we propose a probabilistic, but computationally more efficient approach based on *number theoretic signatures*, which extends recent work by Song et al. [29]. In this approach we compute the signature of a graph as a large, pseudo-unique integer hash that encodes key information such as its vertices, labels, and nodes degree. Graphs with matching signatures are likely to be isomorphic to one another, but there is a small probability of collision, i.e., of having two different graphs with the same signature.

Given a query graph $G_q = \{V_q, E_q\}$ we compute its signature as follows. Initially we assign a random value $r(l) = [1, p)$, between 1 and some user specified prime p , to each possible label $l \in L_{V_i}$ from our data graph G ; recall that the function f_l maps vertices in G to these labels. We then perform three steps:

- (1) Calculate a factor for each edge $e = (v_i, v_j) \in E_q$, according to the formula:

$$edgeFac(e) = (r(f_l(v_i)) - r(f_l(v_j))) \bmod p$$

- (2) Calculate the factors that encode the degree of each vertex. If a vertex v has a degree n , its degree factor is defined as:

$$degFac(v) = ((r(f_l(v)) + 1) \bmod p) \cdot ((r(f_l(v)) + 2) \bmod p) \cdot \dots \cdot ((r(f_l(v)) + n) \bmod p)$$

- (3) Finally, we compute the signature of $G_q = (V_q, E_q)$ as:

$$\left(\prod_{e \in E_i} edgeFac(e) \right) \cdot \left(\prod_{v \in V_i} degFac(v) \right)$$

Note that for the factors of *directed* edges, the random value for the target vertex's label is subtracted from the random value for the source vertex's label (i.e. v_j is the target vertex). For *undirected* edges the ordering of subtraction does not matter, provided it is consistent (e.g. lexicographical).

To illustrate this signature calculation process, consider query q_1 from Fig. 1. Given a p of 11 and random values $r(a) = 3$, $r(b) = 10$ we first calculate the edge factor for an $a-b$ edge: $edgeFac((a, b)) = (3 - 10) \bmod 11 = 7$. As q_1 consists of four $a-b$ edges, its total edge factor is $7^4 = 2401$. Then we calculate the

degree factors ³, starting with a b labelled vertex with degree 2: $\text{degFac}(b) = ((10+1) \bmod 11) \cdot ((10+2) \bmod 11) = 11$, followed by an a labelled vertex also with degree 2: $\text{degFac}(a) = 20$. As there are two of each vertex, with the same degree, the total degree factor is $11^2 \cdot 20^2 = 48400$. The signature of $q_1 = 2401 \cdot 48400 = 116208400$.

This approach is appealing for two reasons. Firstly, since the factors in the signature may be multiplied in any order, a signature for G can be calculated incrementally if the signature of any of its sub-graphs G_i is known, as this is the combined factor due to the additional edges and degree in $G \setminus G_i$. Secondly, the choice of p determines a trade-off between the probability of collisions and the performance of computing signatures. Specifically, note that signatures can be **very** large numbers (thousands of bits) even for small graphs, rendering operations such as remainder costly and slow. A small choice of p reduces signature size, because all the factors are mapped to a finite field [17] ($\text{factor} \bmod p$) between 1 and p , but it increases the likelihood of collision, i.e., the probability of two unrelated factors being equal. We discuss how to improve the performance and accuracy of signatures in Section 2.3.

2.2 Constructing the TPSTry++

Algorithm 1 Recursively add a query graph G_q to a TPSTry++

```

1:  $\text{factors}(e, g) \leftarrow$  degree/edge factors to multiply a graph  $g$ 's
   signature when adding edge  $e$ 
2:  $\text{support}(g) \leftarrow$  map of TPSTry++ nodes (graphs) to  $p$ -values
3:  $\text{tpstry} \leftarrow$  TPSTry++ for workload  $Q$ 
4:  $\text{parent} \leftarrow$  TPSTry++ node, initially root (an empty graph)
5:  $G_q \leftarrow$  query graph defined by a query  $q$ 
6:  $g \leftarrow$  some sub-graph of  $G_q$ 

7: for  $e$  in edges from  $G_q$  do
8:    $g \leftarrow$  new empty graph
9:   corecurse( $\text{parent}, e, \text{tpstry}, g$ )
10:     $\text{sig} \leftarrow \text{factor}(e, g) \cdot \text{parent.signature}$ 
11:    if  $\text{tpstry.signatures}$  contains  $\text{sig}$  then
12:       $n \leftarrow$  node from  $\text{tpstry}$  with signature  $\text{sig}$ 
13:       $\text{support}(n) \leftarrow \text{support}(n) + 1$ 
14:    else
15:       $n \leftarrow$  new node with graph  $g + e$  and signature  $\text{sig}$ 
16:       $\text{support}(n) \leftarrow 1$ 
17:       $\text{tpstry} \leftarrow \text{tpstry} + n$ 
18:    if not  $\text{parent.children}$  contains  $n$  then
19:       $\text{parent.children} \leftarrow \text{parent.children} + n$ 
20:     $\text{newEdges} \leftarrow$  edges incident to  $g + e$  and not in  $g + e$ 
21:    for  $e'$  in  $\text{newEdges}$ 
22:      corecurse( $n, e', \text{tpstry}, g + e$ )
23: return  $\text{tpstry}$ 

```

Our approach to constructing the TPSTry++ is to incrementally compute signatures for sub-graphs of each query graph q in a trie, merging trie nodes with equal signatures to produce a DAG which encodes the sub-graphs of all $q \in Q$. Alg. 1 formalises this approach.

Essentially, we recursively “rebuild” the graph $G_q | E_q |$ times, starting from each edge $e \in E_q$ in turn. For an edge e we calculate its edge and degree factors, initially assuming a degree of 1 for

³Note we don't consider 0 a valid factor, and replace it with p (e.g. $11 \bmod 11 = 11$)

each vertex. If the resulting signature is not associated with a child of the TPSTry++'s root, then we add a node n representing e . Subsequently, we “add” those edges e' which are incident to $e \in G_q$, calculating the additional edge and degree factors, and add corresponding trie nodes as children of n . Then we recurse on the edges incident $e + e'$.

Consider again our earlier example of the query graph q_1 : as it arrives in the workload stream Q , we break it down to its constituent edges $\{a-b, a-b, a-b, a-b\}$. Choosing an edge at random we calculate its combined factor. We know that the edge factor of an $a-b$ edge is 7. When considering this single edge, both a and b vertices have a degree of 1, therefore the signature for $a-b$ is $7 \cdot ((3 + 1) \bmod 11) \cdot ((10 + 1) \bmod 11) = 308$. Subsequently, we do the same for all other edges and, finding that they have the same signature, leave the trie unmodified. Next, for each edge, we add each incident edge from q_1 and compute the new combined signature. Assume we add another $a-b$ edge adjacent to b to produce the sub-graph $a-b-a$. This produces three new factors: the new edge factor 7, the new a vertex degree factor $((3 + 1) \bmod 11)$ and an additional degree factor for the existing b vertex $((10 + 2) \bmod 11)$. The combined signature for $a-b-a$ is therefore $308 \cdot 7 \cdot 4 \cdot 1 = 8624$; if a node with this signature does not exist in the trie as a child of the $a-b$ node, we add it. This continues recursively, considering larger sub-graphs of q_1 until there are no edges left in q_1 which are not in the sub-graph, at which point, q_1 has been added to the TPSTry++.

2.3 Avoiding signature collisions

As mentioned, number theoretic signatures are a probabilistic method of isomorphism checking, prone to collisions. There are several scenarios in which two non-isomorphic graphs may have the same signature: *a*) two factors representing different graph features, such as different edges or vertex degrees, are equal; *b*) two distinct sets of factors have the same product; and *c*) two different graphs have identical sets of edges, vertices and vertex degrees.

The original approach to graph isomorphic checking [29] makes use of an expensive authoritative pattern matching method to verify identified matches. Given a query graph, it calculates its signature in advance, then incrementally computes signatures for sub-graphs which form within a window over a graph stream. If a sub-graph's signature is ever divisible by that of the query graph, then that sub-graph should contain a query match.

There are some key differences in how we compute and use signatures with Loom, which allow us to rely solely upon signatures as an efficient means for mining and matching motifs. Firstly, remember our overall aim is to heuristically lower the probability that sub-graphs in a graph G which match our discovered motifs straddle a partition boundary. As a result we can tolerate some small probability of false positive results, whilst the manner in which signatures are executed (Sec. 2.1) precludes false negatives; i.e. two graphs which **are** isomorphic are guaranteed to have the same signature. Secondly, we can exploit the structure of the TPSTry++ to avoid ever explicitly computing graph signatures. From Fig. 2 and Alg. 1, we can see that all possible sub-graphs of a query graph G_q will exist in the TPSTry++ by construction. We calculate the edge and degree factors which would multiply the signature of a sub-graph with the addition of each edge, then associate these factors to the relevant trie branches. This allows us to represent signatures as sets of their constituent factors,

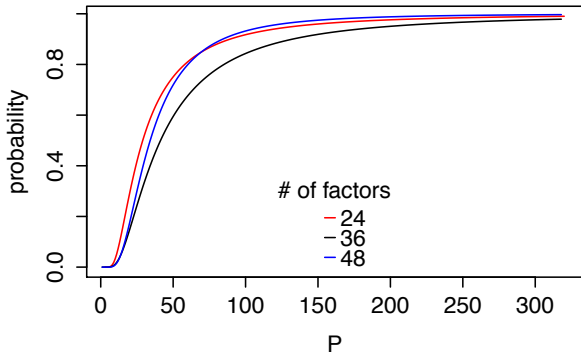


Figure 4: Probability of < 5% factor collisions for various numbers of factors and finite fields p

which eliminates a source of collisions, e.g. we can now distinguish between graphs with factors $\{6, 2\}$, $\{4, 3\}$ and $\{12\}$. Thirdly, we never attempt to discover whether some sub-graph **contains** a match for query q , only whether it **is** a match for q . In other words, the largest graph for which we calculate a signature is the size of the largest query graph $|G_q|$ for all $q \in Q$, which is typically small⁴. This allows us to choose a larger prime p than Song et al. might, as we are less concerned with signature size, reducing the probability of factor collision, another source of false positive signature matches.

Concretely, we wish to select a value of p which minimises the probability that more than some acceptable percentage $C\%$ of a signature’s factors are collisions. From Section 2.1 there are three scenarios in which a factor collision may occur: *a)* two edge factors are equal despite different vertices with different random values from our range $[1, p)$; *b)* an edge factor is equal to a degree factor; and *c)* two degree factors are equal, again despite different vertices. Song et al. show that all factors are uniform random variables from $[1, p)$, therefore each scenario occurs with probability $\frac{1}{p}$.

For either edge or degree factors, from the above it is clear that there are two scenarios in which a collision may occur, giving a collision probability for any given factor of $\frac{2}{p}$. The Handshaking lemma tells us that the total degree of a graph must equal $2|E|$, which means that a graph must have $3|E|$ factors in its signature: one per edge plus one per degree. Combined with the binary measure of “success” (collision / no collision), this suggests a binomial distribution of factor collision probabilities, specifically $Binomial(3|E|, \frac{2}{p})$. Binomial distributions tell us the probability of exactly x “successes” occurring, however we want the probability that no more than $C_{max} = C\% \cdot 3|E|$ factors collide and so must sum over all acceptable outcomes $x \in C_{max}$:

$$\sum_{x=0}^{C_{max}} Pr(X = x) \text{ where } X \sim Binomial(3|E|, \frac{2}{p})$$

Figure 4 shows the probabilities of having fewer than 5% factor collisions given query graphs of 8, 12 or 16 edges and p choices between 2 and 317. In Loom, when identifying and matching motifs, we use a p value of 251, which as you can see gives a negligible probability of significant factor collisions.

⁴Of the order of 10 edges.

3 MATCHING MOTIFS

We have seen how motifs that occur in Q are identified. By construction, motifs represent graph patterns that are frequently traversed during executions of queries in Q . Thus, the sub-graphs of G that match those motifs are expected to be frequently visited together and are therefore best placed within the same partition. In this section we clarify how we discover pattern matches between sub-graphs and motifs, whilst in the next Section we describe the allocation of those sub-graphs to partitions.

Loom operates on a sliding window of configurable size over the stream of edges that make up the growing graph G . The system monitors the connected sub-graphs that form in the stream within the space of the window, efficiently checking for isomorphisms with any known motif each time a sub-graph grows. Upon leaving the window, sub-graphs that match a motif are immediately assigned to a partition, subject to partition balance constraints as explained in Section 4.

Note that this technique introduces a delay, corresponding to the size of the window, between the time edges are submitted to the system and the time they are assigned and made available. In order to allow queries to access the new parts of graph G , Loom views the sliding window *itself* as an extra partition, which we denote P_{temp} . In practice, vertices and edges in the window are accessible in this temporary partition prior to being permanently allocated to their own partition.

To help understand how the matching occurs, note that in the TPSTry++, by construction, *all* ancestors of any node n must represent strict sub-graphs of the graph represented by n itself. Also, note that the support of a node n is the relative frequency with which n ’s sub-graph G_n occurs in Q . As, by definition, each time G_n occurs in Q so do all of *its* sub-graphs, a trie node n must have a support lower than any of its ancestors. This means that if any of the nodes in the trie, including those representing single edges, are not motifs, then none of their descendants can be motifs either. Thus, when a new edge $e = (v_1, v_2)$ arrives in the graph stream, we compute its signature (Sec. 2.1) and check if e matches a single-edge motif at the root of the TPSTry++. If there is no match, we can be certain that e will never form part of any sub-graph that matches a motif. We therefore immediately assign e to a partition and do not add it to our stream window P_{temp} . If, on the other hand, e does match a single-edge motif then we record the match into a map, *matchList*, and add e to the window. The *matchList* maps vertices v to the set of motif matching sub-graphs in P_{temp} which contain v ; i.e. having determined that $e = (v_1, v_2)$ is a motif match, we treat e as a sub-graph of a single edge, then add it to the *matchList* entries for both v_1 and v_2 . Additionally, alongside every sub-graph in *matchList*, we store a reference to the TPSTry++ node which represents the matching motif. Therefore, entries in *matchList* take the form $v \rightarrow \{\langle E_i, m_i \rangle, \langle E_j, m_j \rangle, \dots\}$, where E_i is a set of edges in P_{temp} that form a sub-graph g_i with the same signature as the motif m_i .

Given the above, any edge e which is added to P_{temp} must at least match a single edge motif. However, if e is incident to other edges already in P_{temp} , then its addition may also form larger motif matching sub-graphs which we must also detect and add to *matchList*. Thus, having added $e = (v_1, v_2)$ to *matchList*, we check the map for existing matches which are connected to e ; i.e we look for matches which contain one of v_1 or v_2 . If any exist, we use the procedure in Alg. 2, along with the TPSTry++,

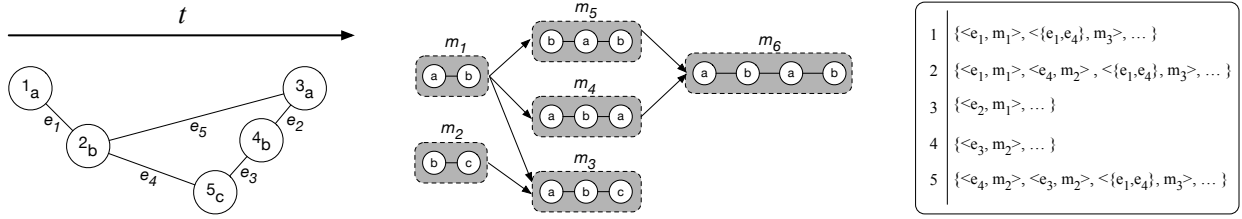


Figure 5: t -length window over G (left), Motifs from TPSTry++ (center) and motif *matchList* for window (right)

Algorithm 2 Mine motif matches from each new edge $e \in G$

- 1: $factors(e, g) \leftarrow$ degree/edge factors to multiply a graph g 's signature when adding edge e
- 2: $tpstry \leftarrow$ filtered TPSTry++ of motifs for workload Q
- 3: **for** each new edge $e(v_1, v_2)$ **do**
- 4: $matches \leftarrow matchList(v_1) \cup matchList(v_2)$
- 5: **for** each sub-graph m in $matches$ **do**
- 6: $n \leftarrow$ the *tpstry* node for m
- 7: **if** n has child c w. $factor = factors(e, m)$ **then**
- 8: add $\langle m + e, c \rangle$ to *matchList* for $v_1 \& v_2$ //Match found!
- 9: $ms_1 \leftarrow matchList(v_1)$
- 10: $ms_2 \leftarrow matchList(v_2)$
- 11: **for** all possible pairs (m_1, m_2) from (ms_1, ms_2) **do**
- 12: $n_1 \leftarrow$ the *tpstry* node for m_1
- 13: **recurse**(*tpstry*, m_2, m_1, n_1)
- 14: **for** each edge e_2 in m_2 **do**
- 15: **if** n_1 has child c_1 w. $factor = factors(e_2, m_1)$ **then**
- 16: **recurse**(*tpstry*, $m_2 - e_2, m_1 + e_2, c_1$)
- 17: **if** m_2 is empty **then** //Match found!
- 18: add $\langle m_1 + m_2, n_1 \rangle$ to *matchList* for $v_1 \& v_2$

to determine whether the addition of edge e to these sub-graphs creates another motif match.

Essentially, for each sub-graph g_i from *matchList* to which e is connected, we calculate the set of edge and degree factors $fac(e, g_i)$ which would multiply the signature of g_i upon the addition e , as in Sec. 2. Recall, also from Sec. 2, that a TPSTry++ node contains a signature for the graph it represents, and that these signatures are stored as sets of factors, rather than their large integer products. As each sub-graph in *matchList* is paired with its associated motif n from the trie, we can efficiently check if n has a child c where $a) c$ is a motif; and $b)$ the difference between n 's factor set and c 's factor set corresponds to factors for the addition of e to g_i , i.e., $fac(e, g_i) = c.signatures \setminus n.signatures$. If such a child exists in the trie then adding e to a graph which matches motif n (g_i) will likely create a graph which matches motif c : the addition of e to P_{temp} has formed the new motif matching sub-graph $g_i + e$.

We also detect if the joining of two existing multi edge motif matches ($\langle E_1, m_1 \rangle, \langle E_2, m_2 \rangle$) forms yet another motif match, in roughly the same manner. First we consider each edge from the smaller motif match (e.g. $e \in E_2$ from $\langle E_2, m_2 \rangle$), checking if the addition of any of these edges to E_1 ⁵ constitutes yet another match; if it does then we add the edge to E_1 and recursively repeat the process until E_2 is empty. If this process **does** exhaust E_2 then $E_1 \cup E_2$ constitute a motif matching sub-graph. Once this process is complete, *matchList* will contain entries for **all** of

the motif matching sub-graphs currently in P_{temp} . Note that as more edges are added to P_{temp} , *matchList* may contain multiple entries for a given vertex where one match is a sub-graph of another, i.e. new motif matches don't replace existing ones.

As an example of the motif matching process, consider the portion of a graph stream (left), motifs (center) and *matchList* (right) depicted in Fig. 5. Our window over the graph stream G is initially empty, with the depicted edges being added in label order (i.e. e_1, e_2, \dots). As the edge e_1 is added, we first compute its signature and verify whether e_1 matches a single-edge motif in the TPSTry++. We can see that, as an a - b labelled edge, the signature for e_1 must match that of motif m_1 , therefore we add e_1 to P_{temp} , and add the entry $\langle e_1, m_1 \rangle$ to *matchList* for both e_1 's vertices 1,2. As e_1 is not yet connected to any other edges in P_{temp} , we do not need to check for the formation of additional motif matches. Subsequently, we perform the exact same process for edge e_2 . When e_3 is added, again we verify that, as a b - c edge, e_3 is a match for the single-edge motif m_3 and so update P_{temp} and *matchList* accordingly. However, e_3 is connected to existing motif matching sub-graphs in P_{temp} therefore the union of *matchList* entries for e_3 's vertices 4,5 (line 4 Alg. 2) returns $\{\langle e_2, m_1 \rangle\}$. As a result, we calculate the factors to multiply e_2 's signature by, when adding e_3 . Remember that when computing signatures, each edge has a factor, as well as each degree. Thus, when adding e_3 to e_2 our *new* factors are an edge factor for a b - c labelled edge, a first degree factor for the vertex labelled c (5) and a second degree factor for the vertex labelled b ⁶ (4) (Sec. 2.1). Subsequently we must check whether the motif for e_2, m_1 , has any child nodes with additional factors consistent with the addition of a b - c edge, which it does: m_3 . This means we have found a new sub-graph in P_{temp} which matches the motif m_3 , and must add $\{\langle e_2, e_3 \rangle, m_3\}$ to the *matchList* entries for vertices 3, 4 and 5. Similarly, the addition of b - c labelled edge e_4 to our graph stream produces the new motif matches $\langle e_4, m_2 \rangle$ and $\langle e_1, e_4 \rangle, m_3$, as can be seen in our example *matchList*.

Finally, the addition of our last edge, e_5 , creates several new motif matches (e.g. $\langle e_1, e_5 \rangle, m_4 \rangle, \langle e_2, e_5 \rangle, m_5$ etc...). In particular, notice that the addition of e_5 creates a match for the motif m_6 , combining the new motif match $\langle e_1, e_5 \rangle, m_4$ with an existing one $\langle e_2, m_1 \rangle$. To understand how we discover these slightly more complex motif matches, consider Alg. 2 from line 11 onwards. First we retrieve the updated *matchList* entries for vertices 2 and 3, including the new motif matches gained by simply adding the single edge e_5 to connected existing motif matches, as above. Next we iterate through all possible pairs of motif matches for both vertices. Given the pair of matches ($\langle e_1, e_5 \rangle, m_4 \rangle, \langle e_2, m_1 \rangle$), we discover that the addition of any edge from the smaller (i.e. e_2) to the larger produces factors which correspond to a child of m_4 in the TPSTry++: m_6 . As e_2 is the only edge in the smaller

⁵Treating E_1 as a sub-graph.

⁶As, with the addition of e_3 , vertex 4 has degree 2.

match, we simply add the match $\langle \{e_1, e_2, e_5\}, m_6 \rangle$ to the *matchList* entries for 1, 2, 3 and 4. In the general case however, we would not add this new match but instead recursively “grow” it with new edges from the smaller match, updating *matchList* only if all edges from the smaller match have been successfully added.

4 ALLOCATING MOTIFS

Following graph stream pattern matching, we are left with a collection of sub-graphs, consisting solely of the most recent t edges in G , which match motifs from Q . As new edges arrive in the graph stream, our window P_{temp} grows to size t and then “slides”, i.e. each new edge added to a full window causes the oldest ($t + 1^{th}$) edge e to be dropped. Our strategy with Loom is to then assign this old edge e to a permanent partition, along with the other edges in the window which form motif matching sub-graphs with e . The sole exception to this is when an edge arrives that may not form part of any motif match and is assigned to a partition immediately (Sec. 3). This exception does not pose a problem however, because Loom behaves as if the edge was never added to the window and therefore does not cause displacement of older edges.

Recall again that with Loom we are attempting to assign motif matching sub-graphs wholly within individual partitions with the aim of reducing *ipt* when executing our query workload Q . One naive approach to achieving this goal is as follows: When assigning an edge $e = (v_1, v_2)$, retrieve the motif matches associated with v_1 and v_2 from P_{temp} using our *matchList* map, then select the subset M_e that contains e , where $M_e = \{\langle E_1, m_1 \rangle, \dots, \langle E_n, m_n \rangle\}$, $e \in E_i$ and E_i is a match for m_i . Finally, treating these matches as a single sub-graph, assign them to the partition which they share the most incident edges. This approach would greedily ensure that no edges belonging to motif matching sub-graphs in G ever cross a partition boundary. However, it would likely also have the effect of creating highly unbalanced partition sizes, potentially straining the resources of a single machine, which prompted partitioning in the first place.

Instead, we rely upon two distinct heuristics for edge assignment, both of which are aware of partition balance. Firstly, for the case of non-motif-matching edges that are assigned immediately, we use the existing *Linear Deterministic Greedy* (LDG) heuristic [30]. Similar to our naive solution above, LDG seeks to assign edges⁷ to the partition where they have the most incident edges. However, LDG also favours partitions with higher residual capacity when assigning edges in order to maintain a balanced number of vertices and edges between each. Specifically, LDG defines the residual capacity r of a partition S_i in terms of the number of vertices currently in S_i , given as $|\mathcal{V}(S_i)|$, and a partition capacity constraint C : $r(S_i) = 1 - \frac{|\mathcal{V}(S_i)|}{C}$. When assigning an edge e , LDG counts the number of e 's incident edges in each partition, given as $N(S_i, e)$, and weights these counts by S_i 's residual capacity; e is assigned to the partition with the highest weighted count. The full formula for LDG's assignment is:

$$\max_{S_i \in P_k(G)} N(S_i, e) \cdot \left(1 - \frac{|\mathcal{V}(S_i)|}{C}\right)$$

Secondly, for the general case where edges form part of motif matching sub-graphs, we propose a novel heuristic, *equal opportunism*. Equal opportunism extends ideas present in LDG but, when assigning clusters of motif matching sub-graphs to a single partition as we do in Loom, it has some key advantages.

By construction, given an edge e to be assigned along with its motif matches $M_e = \{\langle E_1, m_1 \rangle, \dots, \langle E_n, m_n \rangle\}$, the sub-graphs E_i, E_j in M_e have significant overlap (e.g. they all contain e). Thus, individually assigning each motif match to potentially different partitions would create many inter-partition edges. Instead, equal opportunism greedily assigns the match cluster to the single partition with which it shares the most vertices, weighted by each partition's residual capacity. However, as these vertices and their new motif matching edges may not be traversed with equal likelihood given a workload Q , equal opportunism also prioritises the shared vertices which are part of motif matches with higher support in the TPSTry++.

Formally, given the motif matches M_e we compute a score for each partition S_i and motif match $\langle E_k, m_k \rangle \in M_e$, which we call a *bid*. Let $\mathcal{N}(S_i, E_k) = |\mathcal{V}(S_i) \cap \mathcal{V}(E_k)|$ denote the number of vertices in the edge set E_k (which is itself a graph) that are already assigned to S_i ⁸. Additionally, let $supp(m_k)$ refer to the support of motif m_k in the TPSTry++ and recall that C is a capacity constraint defined for each partition. We define the bid for partition S_i and motif match $\langle E_k, m_k \rangle$ as:

$$bid(S_i, \langle E_k, m_k \rangle) = \mathcal{N}(S_i, E_k) \cdot \left(1 - \frac{|\mathcal{V}(S_i)|}{C}\right) \cdot supp(m_k) \quad (1)$$

We could simply assign the cluster of motif matching sub-graphs (i.e. $E_1 \cup \dots \cup E_n$) to the single partition S_i with the highest bid for all motif matches in M_e . However, equal opportunism further improves upon the balance and quality of partitionings produced with this new weighted approach, limiting its greediness using a rationing function we call l . $l(S_i)$ is a number between 0 and 1 for each partition, the size of which is inversely correlated with S_i 's size relative to the smallest partition $S_{min} = \min_{S \in P_k(G)} |\mathcal{V}(S)|$, i.e. if S_i is as small as S_{min} then $l(S_i) = 1$. Equal opportunism sorts motif matches in M_e in descending order of support, then uses $l(S_i)$ to control both the number of matches used to calculate partition S_i 's total bid, and the number of matches assigned to S_i should its total bid be the highest. This strategy helps create a balanced partitioning by a) allowing smaller partitions to compute larger total bids over more motif matches; and b) preventing the assignment of large clusters of motif matches to an already large partition. Formally we calculate $l(S_i)$ as follows:

$$l(S_i) = \frac{|\mathcal{V}(S_i)|}{S_{min}} \cdot \alpha, \quad \alpha = \begin{cases} 1, & |\mathcal{V}(S_i)| = |\mathcal{V}(S_{min})| \\ 0, & |\mathcal{V}(S_i)| > |\mathcal{V}(S_{min})| \cdot b \\ \alpha, & \text{otherwise} \end{cases} \quad (2)$$

where α is a user specified number $0 < \alpha \leq 1$ which controls the aggression with which l penalises larger partitions and b limits the maximum imbalance. Throughout this work we use an empirically chosen default of $\alpha = \frac{2}{3}$ and set the maximum imbalance to $b = 1.1$, emulating Fennel [31].

Given definitions (1) and (2), we can now simply state the output of equal opportunism for the sorted set of motif matches M_e , as:

$$\max_{S_i \in P_k(G)} l(S_i) \cdot |M_e| \sum_{k=0} bid(S_i, \langle E_k, m_k \rangle) \quad (3)$$

Note that motif matches in M_e which are not bid on by the winning partition are dropped from the *matchList* map, as some of their constituent edges (e.g. e , which all matches in M_e share) have been assigned to partitions and removed from the sliding window P_{temp} .

⁷LDG may partition either vertex or edge streams.

⁸Note that \mathcal{N} is a generalisation of LDG's function \mathcal{N}

To understand how the rationing function l improves the quality of equal opportunism’s partitioning, not just its balance, consider the following: Just because an edge e' falls within the motif match set M_e of our assignee e , does not necessarily imply that placing them within the same partition is optimal. e' could be a member of many other motif matches in P_{temp} besides those in M_e , perhaps with higher support in the TPSTry++ (i.e. higher likelihood of being traversed when executing a workload Q). By ordering matches by support and prioritising the assignment of the smaller, higher support motif matches, we often leave e' to be assigned later along with matches to which it is more “important”.

As an example, consider again the graph and TPSTry++ fragment in Fig. 5. If assigning the edge e_1 to a partition at the time $t + 1$, its **support ordered** set of motif matches M_{e_1} would be $\langle e_1, m_1 \rangle, \langle \{e_1, e_4\}, m_3 \rangle, \langle \{e_1, e_5\}, m_4 \rangle$ and $\langle \{e_1, e_2, e_5\}, m_6 \rangle$. Assume two partitions S_1 and S_2 , where S_1 is 33.3% larger than S_2 and vertex 2 already belongs to partition S_1 , whilst all other vertices in the window are as yet unassigned (i.e. this is the first time edges containing them have entered the sliding window). In this scenario, S_1 is guaranteed to win all bids, as S_2 contains no vertices from M_{e_1} and therefore $\mathcal{N}(S_2, _)$ will always equal 0. However, rather than greedily assign all matches to the already large S_1 , we calculate the ration l for S_1 as $\frac{1}{1.33} \cdot \frac{1}{1.5} = \frac{1}{2}$, given $\alpha = 1.5$. In other words, we only assign edges from the first half of M_{e_1} ($\langle e_1, m_1 \rangle, \langle \{e_1, e_4\}, m_3 \rangle$) to S_1 ; edges such as e_5 and e_2 remain in the window P_{temp} . Assume an edge $e_6 = (4, 6)$ subsequently arrives in the graph stream G , where vertex 6 already belongs to partition S_2 and e_6 matches the motif m_2 (i.e. has labels $b-c$). If we had already assigned e_5 to partition S_1 then this would lead to an inter-partition edge which is more likely to be traversed together with e_5 than are other edges in S_1 , given our workload Q . Instead, we compute a match in P_{temp} between $\langle e_5, e_6 \rangle$ and the motif m_3 , and will likely later assign e_5 to partition S_2 . Within reason, the longer an edge remains in the sliding window, the more of its neighbourhood information we are likely to have access to, the better partitioning decisions we can make for it.

5 EVALUATION

Our evaluation aims to demonstrate that Loom achieves high quality partitionings of several large graphs in a single-pass, streaming manner. Recall that we measure graph partitioning quality using the number of inter-partition traversals when executing a realistic workloads of pattern matching queries over each graph.

Loom consistently produces partitionings of around 20% superior quality when compared to those produced by state of the art alternatives: LDG [30] and Fennel [31] Furthermore, Loom partitionings’ quality improvement is robust across different numbers of partitions (i.e. a 2-way or a 32-way partitioning). Finally we show that, like other streaming partitioners, Loom is sensitive to the arrival order of a graph stream, but performs well given a pseudo-adversarial random ordering.

5.1 Experimental setup

For each of our experiments, we start by streaming a graph from disk in one of three predefined orders: **Breadth-first**: computed by performing a breadth-first search across all the connected components of a graph; **Random**: computed by randomly permuting the existing order of a graph’s elements; and **Depth-first**: computed by performing a depth-first search across the connected

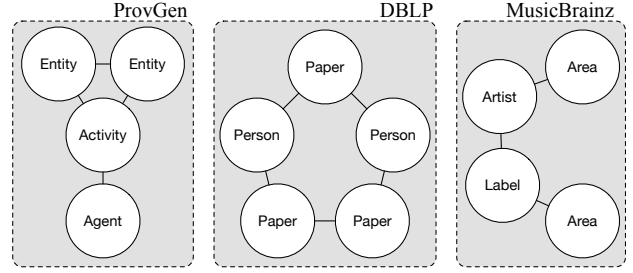


Figure 6: Examples of q for MusicBrainz, DBLP & ProvGen

components of a graph. We choose these stream orderings as they are common to the evaluations of other graph stream partitioners [11, 22, 30, 31], including LDG and Fennel.

Subsequently, we produce 4 separate k -way partitionings of this ordered graph stream, using each of the following partitioning approaches for comparison: **Hash**: a naive partitioner which assigns vertices and edges to partitions on the basis of a hash function. As this is the default partitioner used by many existing partition graph databases⁹, we use it as a baseline for our comparisons. **LDG**: a simple graph stream partitioner with good performance which we extend with our work on Loom. **Fennel**: a state-of-the-art graph stream partitioner and our primary point of comparison. As suggested by Tsourakakis et al, we use the Fennel parameter value $\gamma = 1.5$ throughout our evaluation. **Loom**: our own partitioner which, unless otherwise stated, we invoke with a window size of 10k edges and a motif support threshold of 40%.

Finally, when each graph is finished being partitioned, we execute the appropriate query workload over it and count the number of inter-partition traversals (ipt) which occur.

Note that we avoid implementation dependent measures of partitioning quality because, as an isolated prototype, Loom is unlikely to exhibit realistic performance. For instance, lacking a distributed query processing engine, query workloads are executed over logical partitions during the evaluation. In the absence of network latency, query response times are meaningless as a measure of partitioning quality.

All algorithms, data structures, datasets and query workloads are publicly available¹⁰. All our experiments are performed on a commodity machine with a 3.1Ghz Intel i7 CPU and 16GB of RAM.

5.1.1 Graph datasets. Remember that the workload-agnostic partitioners which we aim to supersede with Loom are liable to exhibit poor workload performance when queries focus on traversing a limited subset of edge types (Sec. 1). Intuitively, such skewed workloads are more likely over heterogeneous graphs, where there exist a larger number of possible edge types for queries to discern between, e.g. $a-a$, $a-b$, $a-c$... vs just $a-a$. Thus, we have chosen to test the Loom partitioner over five datasets with a range of different heterogeneities and sizes; three of these datasets are synthetic and two are real-world. Table 1 presents information about each of our chosen datasets, including their size and how heterogeneous they are ($|L_V|$). We use the DBLP, and LUBM datasets, which are well known. MusicBrainz¹¹ is a freely available database of curated music metadata, with vertex

⁹The Titan graph database: <http://bit.ly/2ejypXV>

¹⁰The Loom repository: <http://bit.ly/2eJxQcp>

¹¹The MusicBrainz database: <http://bit.ly/1J0wlNR>

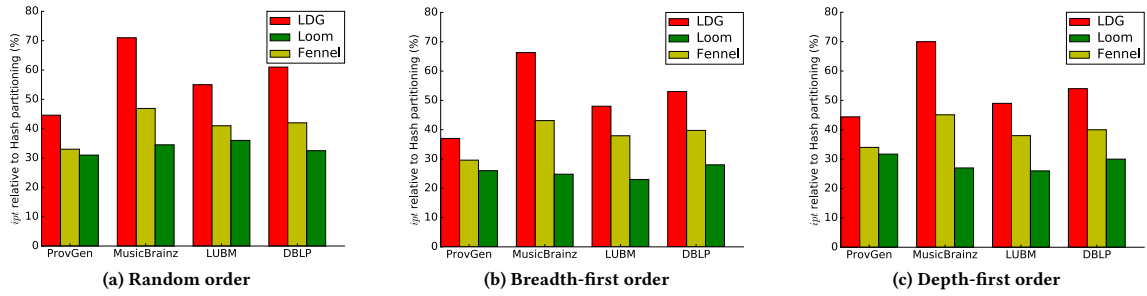


Figure 7: ipt %, vs. Hash, when executing Q over 8-way partitionings of graph streams in multiple orders.

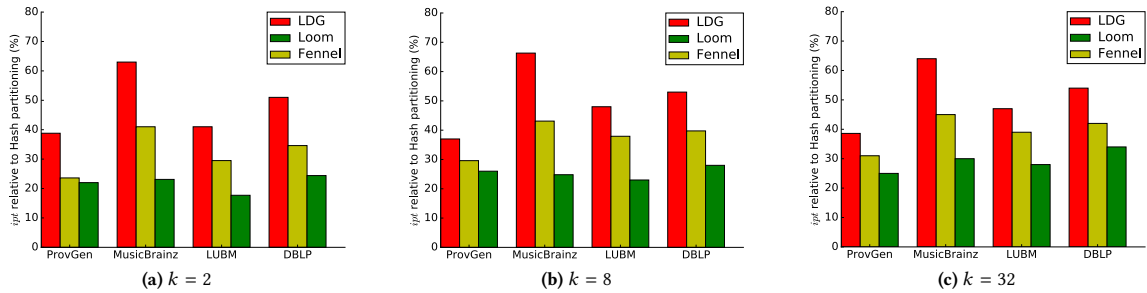


Figure 8: ipt %, vs. Hash, when executing Q over multiple k -way partitionings of breadth-first graph streams.

Dataset	$\sim V $	$\sim E $	$ L_V $	Real	Description
DBLP	1.2M	2.5M	8	Y	Publications & citations
ProvGen	0.5M	0.9M	3	N	Wiki page provenance
MusicBrainz	31M	100M	12	Y	Music records metadata
LUBM-100	2.6M	11M	15	N	University records
LUBM-4000	131M	534M	15	N	University records

Table 1: Graph datasets, incl. size & heterogeneity

labels such as *Artist*, *Country*, *Album* and *Label*. ProvGen[6] is a synthetic generator for PROV metadata [21], which records detailed provenance for digital artifacts.

5.1.2 Query workloads. For each dataset we must propose a representative query workload to execute so that we may measure partitioning quality in terms of ipt . Remember that a query workload consists of a set of distinct query patterns along with a frequency for each (Sec. 1.3). The LUBM dataset provides a set of query patterns which we make use of. For every other dataset, however, we define a small set of common-sense queries which focus on discovering implicit relationships in the graph, such as potential collaboration between authors or artists¹². The full details of these query patterns are elided for space¹⁰, however Fig. 6 presents some examples. Note that whilst the TPSTry++ may be trivially updated to account for change in the frequencies of workload queries (Sec. 2), our evaluation of Loom assumes that said frequencies are fixed and known *a priori*. Recall that, for online databases, we argue this is a realistic assumption (Sec. 1). However, more complete tests with changing workloads are an important area for future work.

¹²If possible, workloads are drawn from the literature, e.g. common PROV queries [5]

5.2 Comparison of systems

Figures 7 and 8 present the improvement in partitioning quality achieved by Loom and each of the comparable systems we describe above. Initially, consider the experiment depicted in Fig. 7. We partition ordered streams of each of our first 4 graph datasets¹³ into 8-way partitionings, using the approaches described above, then execute each dataset’s query workload over the appropriate partitioning. The absolute number of inter-partition traversals (ipt) suffered when querying each dataset varies significantly. Thus, rather than represent these results directly, in Fig. 7 (and 8) we present the results for each approach as **relative** to the results for Hash; i.e. how many ipt did a partitioning suffer, as a **percentage** of those suffered by the Hash partitioning of the same dataset.

As expected, the naive hash partitioner performs poorly: it produces partitionings which suffer twice as many inter-partition traversals, on average, when compared to partitionings produced by the next best system (LDG). Whilst the LDG partitioner does achieve around a 55% reduction in ipt vs our Hash baseline, its produces partitionings of consistently poorer quality than those of Fennel and Loom. Although both LDG and Fennel optimise their partitionings for the balanced min. edge-cut goal (Sec. 1), Fennel is the more effective heuristic, cutting around 25% fewer edges than LDG for small numbers of partitions (including $k = 8$) [31]. Intuitively, the likelihood of *any* edge being cut is a coarse proxy for the likelihood of a query $q \in Q$ traversing a cut edge. This explains the disparity in ipt scores between the two systems.

Of more interest is comparing the quality of partitionings produced by Fennel and Loom. Fig. 7 clearly demonstrates that Loom offers a significant improvement in partitioning quality over Fennel, given a workload Q . Loom’s reduction in ipt relative to Fennel’s is present across all datasets and stream orders, however

¹³Excluding LUBM-4000

Dataset	LDG (ms)	Fennel (ms)	Loom (ms)	Hash (ms)
DBLP	91	96	235	28
ProvGen	144	146	240	33
MusicBrainz	48	52	129	18
LUBM-100	47	51	147	22
LUBM-4000	45	49	138	16

Table 2: Time to partition 10k edges

it is particularly pronounced over ordered streams of more heterogeneous graphs; e.g. MusicBrainz in Sub-figure 8b(b), where Loom’s partitioning suffers from 42% fewer *ipt* than Fennel’s. This makes sense because, as mentioned, pattern matching workloads are more likely to exhibit skew over heterogeneous graphs, where query graphs G_q contain a, potentially small, subset of the possible vertex labels. Across all the experiments presented in Fig. 7, the median range of Loom’s *ipt* reduction relative to Fennel’s is 20 – 25%. Additionally, Fig. 8 demonstrates that this improvement is consistent for different numbers of partitions. As the number of partitions k grows, there is a higher probability that vertices belonging to a motif match are assigned across multiple partitions. This results in an increase of *absolute ipt* when executing Q over a Loom partitioning. However, increasing k actually increases the probability that any two vertices which share an edge are split between partitions, thus reducing the quality of Hash, LDG and Fennel partitionings as well. As a result, the difference in *relative ipt* is largely consistent between all 4 systems.

On the other hand, neither Fig. 7, nor Fig. 8, present the runtime costs of producing a partitioning. Table 2 presents how long (in ms) each partitioner takes to partition 10k edges. Whilst all 3 algorithms are capable of partitioning many 10s of thousands of edges per second, we do find that Loom is slower than LDG and Fennel by an average factor of 2-3. This is likely due to the more complex map-lookup and pattern-matching logic performed by Loom, or a nascent implementation. The runtime performance of Loom varies depending on the query workload Q used to generate the TPSTry++ (Sec. 2), therefore the performance figures presented in Table 2 are averaged across many different Q . The minimum slowdown factor observed between Loom and Fennel was 1.5, the maximum 7.1. Note that popular non-streaming partitioner METIS [14] is around 13 times slower than Fennel for large graphs [31].

We contend that this performance difference is unlikely to be an issue in an online setting for two reasons. Firstly, most production databases do not support more than around 10k transactions per second (TPS) [16]. Secondly, it is considered exceptional for even applications such as twitter to experience >30k-40k TPS¹⁴. Meanwhile, the lowest partitioning rate exhibited by Loom in Table 2 is equivalent to ~ 42k edges per second, the highest 72k.

Note that Figures 7 and 8 do not present the relative *ipt* figures for the LUBM-4000 dataset. This is because measuring relative *ipt* involves reading a partitioned graph into memory, which is beyond the constraints of our present experimental setup. However, we include the LUBM-4000 dataset in Table 2 to demonstrate that, as a streaming system, Loom is capable of partitioning large scale graphs. Also note that none of the figures present partitioning imbalance as this is broadly similar between all approaches and

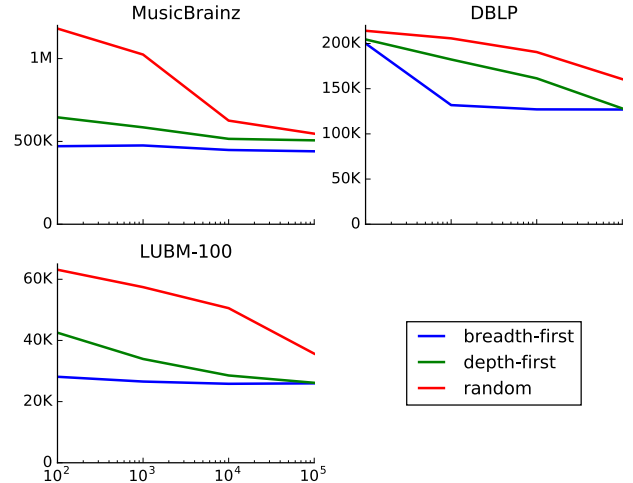


Figure 9: *ipt* (y-axis) when executing Q over Loom partitionings with multiple window sizes t (x-axis)

datasets¹⁵, with LDG varying between 1%–3%, Loom and Fennel between 7% and their maximum imbalance of 10% (Sec. 4).

5.3 Effect of stream order and window size

Fig. 7 indicates that Loom is sensitive to the ordering of its given graph stream. In fact, Sub-figure 7(a) shows Loom achieve a smaller reduction in *ipt* over Fennel and LDG, than in 7(b) and 7(c). Specifically Loom achieves a 42% greater reduction in relative *ipt* than Fennel given a breadth-first stream of the MusicBrainz graph, but only a 26% when the stream is ordered randomly, despite Fennel and LDG also being sensitive to stream ordering [30, 31]. This implies that Loom is particularly sensitive to random orderings: edges which are close to one another in the graph may not be close in the graph stream, resulting in Loom detecting fewer motif matching subgraphs in its stream window.

Intuitively, this sensitivity can be ameliorated by increasing the size of Loom’s window, as shown in Fig. 9. As Loom’s window grows, so does the probability that clusters of motif matching subgraphs will occur within it. This allows Loom’s equal opportunism heuristic to make the best possible allocation decisions for the subgraph’s constituent vertices. Indeed, the number of *ipt* suffered by Loom partitionings improves significantly, by as much as 47%, as the window size grows from 100 to 10k. However, increasing the window size past 10k clearly has little effect on *ipt* suffered to execute Q if your graph stream is ordered. The exact impact of increasing Loom’s window size depends upon the degree distribution of the graph being partitioned. However, to gain an intuition consider the naive case of a graph with a uniform average vertex degree of 8, along with a TPSTry++ whose largest motif contains 4 edges. In this case, a breadth-first traversal of 8^4 edges from a vertex a (i.e. window size $t \approx 4k$) is highly likely to include all the motif matches which contain a . Regardless, Fig. 9 might seem to suggest that Loom should run with the largest window size possible. However, besides the additional computational cost of detecting more motif matches, remember that Loom’s window constitutes a temporary partition (Sec. 3). If there exist many edges between other partitions and P_{temp} , then this may itself be a source of *ipt* and poor query performance.

¹⁴Tweets per second in 2013: <http://bit.ly/2hQH5JJ>

¹⁵Except Hash, which is balanced.

6 CONCLUSIONS AND FUTURE WORK

In this paper, we have presented Loom: a practical system for producing k -way partitionings of online, dynamic graphs, which are optimised for a given workload of pattern matching queries Q . Our experiments indicate that Loom significantly reduces the number of inter-partition traversals (ipt) required when executing Q over its partitionings, relative to state of the art (workload agnostic) streaming partitioners.

There are several ways in which we intend to expand our current work on Loom. In particular, as a workload sensitive technique, Loom generates partitionings which are vulnerable to workload change over time. In order to address this we must integrate Loom with an existing, workload sensitive, graph repartitioner [8, 10] or consider some form of restreaming approach [11]. In addition to the query workloads already considered, it would be necessary to evaluate such an integrated approach using a dynamic, changing query workload.

Furthermore, due to our approaches reliance upon graph pattern matching in a single stream window, Loom is single threaded. The ability to have multiple instances of the Loom algorithm assign motif matches to the same graph partitioning would doubtless increase system scalability, and is therefore an important focus of ongoing research.

REFERENCES

- [1] K. Andreev and H. Racke. 2006. Balanced Graph Partitioning. *Theory of Computing Systems* 39, 6 (2006), 929–939.
- [2] C. Chevalier and F. Pellegrini. 2008. PT-Scotch: A tool for efficient parallel graph ordering. *Parallel Comput.* 34, 6-8 (2008), 318–331.
- [3] S. Choudhury, L. Holder, G. Chin, et al. 2015. A Selectivity based approach to Continuous Pattern Detection in Streaming Graphs. *In Proc. EDBT (2015)*, 157–168.
- [4] C. Curino, E. Jones, Y. Zhang, et al. 2010. Schism. *In Proc. VLDB* 3, 1-2 (2010), 48–57.
- [5] S. Dey, V. Cuevas-Vicenttin, S. Köhler, et al. 2013. On implementing provenance-aware regular path queries with relational query engines. *In Proc. EDBT/ICDT Workshops*. 214–223.
- [6] H. Firth and P. Missier. 2014. ProvGen: Generating Synthetic PROV Graphs with Predictable Structure. *In Proc. IPAW*. 16–27.
- [7] Hugo Firth and Paolo Missier. 2016. Workload-aware Streaming Graph Partitioning.. *In Proc. EDBT/ICDT Workshops*.
- [8] H. Firth and P. Missier. 2017. TAPER: query-aware, partition-enhancement for large, heterogenous graphs. *Distributed and Parallel Databases* 35, 2 (2017), 85–115.
- [9] P. Gupta, V. Satuluri, A. Grewal, et al. 2014. Real-time twitter recommendation. *In Proc. VLDB* 7, 13 (2014), 1379–1380.
- [10] Razen Harbi, Ibrahim Abdelaziz, Panos Kalnis, Nikos Mamoulis, Yasser Ebrahim, and Majed Sahli. 2016. Accelerating SPARQL Queries by Exploiting Hash-based Locality and Adaptive Partitioning. *The VLDB Journal* 25, 3 (June 2016), 355–380. <https://doi.org/10.1007/s00778-016-0420-y>
- [11] J. Huang and D. Abadi. 2016. LEOPARD : Lightweight Edge-Oriented Partitioning and Replication for Dynamic Graphs. *In Proc. VLDB* 9, 7 (2016), 540–551.
- [12] C. Jiang, F. Coenen, and M. Zito. 2004. A Survey of Frequent Subgraph Mining Algorithms. *The Knowledge Engineering Review* 000 (2004), 1–31.
- [13] A. Jindal and J. Dittrich. 2012. Relax and let the database do the partitioning online. *In Enabling Real-Time Business Intelligence*. 65–80.
- [14] G. Karypis and V. Kumar. 1997. Multilevel k -way Partitioning Scheme for Irregular Graphs. *J. Parallel and Distrib. Comput.* 47, 2 (1997), 109–124.
- [15] B. Kernighan and S. Lin. 1970. An efficient heuristic procedure for partitioning graphs. *Bell systems technical journal* 49, 2 (1970), 291–307.
- [16] S. Lee, B. Moon, C. Park, et al. 2008. A case for flash memory ssd in enterprise database applications. *In Proc. SIGMOD*. 1075.
- [17] R. Lidl and H. Niederreiter. 1997. Finite Fields. *Encyclopedia of Mathematics and Its Applications* (1997), 1983.
- [18] D. Margo and M. Seltzer. 2015. A scalable distributed graph partitioner. *In Proc. VLDB* 8, 12 (2015), 1478–1489.
- [19] B. McKay. 1981. Practical graph isomorphism. (1981), 45–87 pages.
- [20] A. Mendelzon and P. Wood. 1995. Finding Regular Simple Paths in Graph Databases. *SIAM J. Comput.* 24, 6 (1995), 1235–1258.
- [21] L. Moreau, P. Missier, K. Belhajjame, et al. 2012. *PROV-DM: The PROV Data Model*. Technical Report. World Wide Web Consortium.
- [22] J. Nishimura and J. Ugander. 2013. Restreaming graph partitioning. *In Proc. SIGKDD*. New York, New York, USA, 1106–1114.
- [23] A. Pavlo, C. Curino, and S. Zdonik. 2012. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. *In Proc. SIGMOD*. 61.
- [24] P. Peng, L. Zou, L. Chen, et al. 2016. Query Workload-based RDF Graph Fragmentation and Allocation. *In Proc. EDBT*. 377–388.
- [25] Josep M Pujol, Vijay Erramilli, Georgos Siganos, Xiaoyuan Yang, Nikos Laoutaris, Parminder Chhabra, and Pablo Rodriguez. 2010. The little engine(s) that could. *In Proc. SIGCOMM*. 375–386.
- [26] A. Quamar, K. Kumar, and A. Deshpande. 2013. SWORD. *In Proc. EDBT*. 430.
- [27] P. Ribeiro and F. Silva. 2014. G-Tries: a data structure for storing and finding subgraphs. *Data Mining and Knowledge Discovery* 28, 2 (2014), 337–377.
- [28] Z. Shang and J. Yu. 2013. Catch the Wind: Graph workload balancing on cloud. *In Proc. International Conference on Data Engineering (ICDE) (2013)*, 553–564.
- [29] C. Song, T. Ge, C. Chen, and J. Wang. 2014. Event pattern matching over graph streams. *In Proc. VLDB* 8, 4 (2014), 413–424.
- [30] I. Stanton and G. Kliot. 2012. Streaming graph partitioning for large distributed graphs. *In Proc. SIGKDD*. 1222–1230.
- [31] C. Tsourakakis, C. Gkantsidis, B. Radunovic, et al. 2014. FENNEL. *In Proc. ACM International conference on Web search and data mining (WSDM)*. 333–342.
- [32] N. Xu, L. Chen, and B. Cui. 2014. LogGP. *In Proc. VLDB* 7, 14 (2014), 1917–1928.

Kernel-Based Cardinality Estimation on Metric Data

Michael Mattig Thomas Fober Christian Beilschmidt Bernhard Seeger

Department of Mathematics and Computer Science

University of Marburg, Germany

{mattig, thomas, beilschmidt, seeger}@mathematik.uni-marburg.de

ABSTRACT

The efficient management of metric data is extremely important in many challenging applications as they occur e.g. in the life sciences. Here, data typically cannot be represented in a vector space. Instead, a distance function only allows comparing individual elements with each other to support distance queries. As high-dimensional data suffers strongly from the curse of dimensionality, distance-based techniques also allow for better handling of such data. This has already led to the development of a plethora of metric indexing and processing techniques. So far, the important problem of cardinality estimation on metric data has not been addressed in the literature. Standard vector-based techniques like histograms require an expensive and error-prone embedding. Thus, random sampling seems to be the best choice for selectivity estimation so far, but errors are very high for moderately small queries. In this paper, we present a native cardinality estimation technique for distance queries on metric data based on kernel-density estimation. The basic idea is to apply kernels to the one-dimensional distance function among metric objects and to use novel global and local bandwidth optimization methods. Our results on real-world data sets show the clear advantage of our method in comparison to its competitors.

1 INTRODUCTION

Statistics about the distribution of data in a database are used for two very important aspects of data management: query optimization and data exploration. In query optimization, they allow estimating the costs of operations, choosing appropriate algorithms, and computing the order of joins. For very large databases, where computations take a very long time, small in-memory statistics can deliver approximate answers. Those are often sufficient to determine whether it is worth further investigating the data in a particular direction.

While one- and multidimensional vector data is very common in traditional applications, there are many domains for which data is in a metric space only. This means data is not describable by a d -dimensional vector, instead there exists only a metric measuring distances between pairs of objects. Examples include the life sciences, where e.g. proteins are usually described by their geometrical structure or at least a sequence of amino acids. Multimedia data comes in different datatypes such as JPEG or MPEG which are also not appropriate for a relational representation.

In such domains there is a severe lack of native statistical support. Thus, a standard approach is to transform metric data into a multidimensional vector space and to apply one of the standard estimation techniques [18]. There are two serious opposing effects. First, a metric embedding causes in general a considerable information loss. In order to alleviate this, the number of dimensions needs to be sufficiently high. Second, the well-known curse

of dimensionality is already noticeable for a moderate number of dimensions. Thus, statistics provide only accurate results for low-dimensional vector spaces [1].

In this paper, we present the first native method for cardinality estimation of distance queries in metric spaces. The basic idea is to consider the distances of objects in a metric space and to use kernel techniques to estimate the underlying distance distribution. By tuning the bandwidth of the kernels and the kernel function, we obtain a robust estimator for the cardinality of distance queries in metric spaces. Moreover, our approach is also beneficial for high-dimensional vector spaces by treating them as metric spaces, thus considering only the distance among objects, to overcome the shortcomings of standard vectorial statistics.

The main contributions of this paper are:

- We show the deficiencies of traditional cardinality estimation techniques on metric data sets.
- We present the first effective and efficient method for cardinality estimation in metric space.
- Extensive experiments on real-world data show the validity of our approach.

The rest of the paper is structured as follows. Section 2 describes several applications for cardinality estimation in metric spaces, formally defines the problem, and emphasizes the differences of vector and metric data. Section 3 presents related work in the areas of cardinality estimation in general, techniques for embedding metric data into vector space and kernel-based techniques for cardinality estimation. Section 4 presents our distance-based kernel estimator approach in metric space. Section 5 describes our methods for global and local bandwidth optimization. Section 6 presents our experimental findings. Finally, Section 7 concludes the paper.

2 PRELIMINARIES

We first give a formal description of the problem of cardinality estimation on metric data. Then, we discuss several applications that greatly benefit from a suitable solution to this problem. Finally, we discuss the fundamental differences between vector and metric data that lead to the ineffectiveness of established methods.

2.1 Problem Specification

Let X be a set of N objects $\{x_1, \dots, x_N\} \subseteq \mathcal{X}$. These objects are all of a certain type, in particular a type which can differ from \mathbb{R}^n . Moreover a distance function $dist_{\mathcal{X}} : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}_+$ is given which fulfills the three properties of a metric, namely

- identity of indiscernibles: $dist_{\mathcal{X}}(x, y) = 0 \Leftrightarrow x = y$,
- symmetry: $dist_{\mathcal{X}}(x, y) = dist_{\mathcal{X}}(y, x)$ and
- triangle inequality: $dist_{\mathcal{X}}(x, z) \leq dist_{\mathcal{X}}(x, y) + dist_{\mathcal{X}}(y, z)$,

with $x, y, z \in \mathcal{X}$. We will refer to the combination of X and $dist_{\mathcal{X}}$ as metric data. In mathematics the pair $(\mathcal{X}, dist_{\mathcal{X}})$ is called metric space.

Cardinality estimation for metric data can be formalized as follows: Given a distance query $Q = (x_Q, r_Q)$, with object $x_Q \in \mathcal{X}$

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

and distance $r_Q \in \mathbb{R}_+$, *efficiently* approximate the cardinality of the set $\{x \in X \mid \text{dist}_X(x_Q, x) \leq r_Q\}$. We will refer to the distance r_Q as query radius. The true cardinality is denoted by $c(Q)$ and the estimated cardinality by $\tilde{c}(Q)$. The goal is to minimize the error of the estimation, but also the construction costs and the size, as well as the query time of the estimator. Note that the actual cardinality can, of course, be calculated by computing the distance to every other item in the data set. This requires a linear number of distance calculations. Each of them can be very costly as, e.g., in video dissimilarity. Hence, we want to minimize the computational costs induced by an estimator.

2.2 Applications

Cardinality estimation in metric spaces has many applications. Among others we want to mention the domain of Machine Learning and Data Mining, where algorithms are usually based on a distance measure. The most prominent example is the so-called k -nearest neighbor (k NN) classifier which can be used to classify any kind of data, if it is endowed with a distance measure.

The basic idea is to retrieve the k objects from a database which have the smallest distances to a certain query object. Assuming that the objects within the database carry a certain class label (e.g. customers of an insurance with label *churn* or *no churn*), the query is classified with the majority label from the set of the k nearest neighbors. k NN classifiers are known to be inefficient since for each run of such an algorithm a complete scan of the data set is required. To accelerate this algorithm typically metric index structures [28] are employed that allow the efficient retrieval of elements within a given distance of a query object. However, it is hard to specify the radius for the corresponding queries since the k NN classifier requires rather the set of k nearest neighbors than a certain set of neighbors exhibiting a certain maximal distance to the query object. For calculating the minimum radius, leading to a retrieval of k results, cardinality estimation can be used. For example, [14] make use of cardinality estimation to assign objects to different Locality Sensitive Hashing tables of different radii. This improves k NN queries on data sets where the distances of the k nearest neighbours of items vary greatly over the data set. A single LSH table could not sufficiently answer such queries.

Another example is the estimation of densities, e.g. for Bayes Classifiers, where the density around an element x is proportional to the amount of elements in a database that are in a small vicinity to x . This vicinity is typically specified by a certain small distance. Obviously, a reliable cardinality estimation approach would increase the efficiency of such classifiers enormously.

If a query is expressed as a conjunction of multiple proximity predicates, each of them with a different distance function, cardinality estimation is useful for computing an efficient order of their computation. In an optimal execution plan queries should be applied in an order that leads to quickly decreasing result sets. Cardinality estimation can be used to answer exactly this question and to find an order in which the different distance measures are to be applied. Applications in which such scenarios occur are, e.g., pharmaceutical chemistry, where different distance measures covering certain requirements are applied onto protein and/or ligand databases to get the final result in form of a very small set of therapeutically effective drugs. In general, this is a metric scenario, since proteins cannot be described on the structural level by vectors without a considerable loss of information.

2.3 Vector Data vs. Metric Data

In order to emphasize the fundamental differences of metric data to vector data that lead to the in-applicability of established methods, we now briefly review important properties of a vector space. We limit our discussion to vector spaces over the real numbers. Here, a d -dimensional vector space consists of elements $\mathbf{x} = (x_1, \dots, x_d)$ with a real value $x_i \in \mathbb{R}$ called *coordinate* for each dimension. Individual elements can be added to each other and multiplied with scalar values $v \in \mathbb{R}$. This, e.g., allows to compute the mean of multiple elements which is not possible in a metric space. Thus one of the most basic data summarization operators is not available in a metric space. Furthermore, the coordinates of the vector allow determining the location of an element with respect to other elements. Such a *direction* cannot be determined in a metric space.

A set of vectors can be ordered globally by component-wise sorting or by a space-filling curve [27] that better preserves the proximity of subsequent elements. In contrast, elements in a metric space can only be ordered based on the distance to a single reference object. Furthermore, it is straight-forward to divide a vector space into a finite number of distinct subsets by incrementally subdividing the space along the dimensions. In a metric space such subsets have to be defined using a center object and a radius. In general, such partitions will overlap if the complete data space should be covered.

A vector space has a *measure* that allows calculating the volume of subspaces and their intersections. In particular, this is the foundation for the definition of a density and a distribution of a data set. The notions of volume, density and distribution are not available in a metric space.

Finally, in a vector space, the costs of distance calculations between elements is linear in the number of dimension if an L_p norm (typically $p = 2$ for Euclidean distance) is used. In a metric space a distance function can be arbitrarily complex, such as e.g. the edit distance between two strings which has a quadratic runtime. Furthermore, we can calculate a bounding box of a vector data set in linear time by finding the minimum and maximum value for each dimension. In contrast, finding the maximum distance between elements in a metric space requires a quadratic number of distance computations. In summary, metric data lacks most of the tools available in traditional scenarios for cardinality estimation. This makes most established methods infeasible as we discuss in the Section 3.

However, as discussed previously, metric data appears in many different applications naturally. Furthermore, it supports distance queries, which are also highly relevant for vector data [4]. As our experiments will show later, using distance-based techniques helps lowering the impact of the curse of dimensionality.

3 RELATED WORK

The most basic idea for estimating the size of a query result is to perform the query on a sample of the data and scale up the resulting cardinality by the sample's fraction of the total data size. Using Reservoir Sampling [32], a random data selection can be computed in linear time. We can apply this method also on metric data. However, small sample sizes result in underestimates often equal to zero because metric spaces are sparse.

Histograms are the most popular technique for cardinality estimation in database systems [18]. They divide a domain into multiple buckets and store the number of contained elements. When estimating the cardinality within a given query range, they

approximate the actual cardinality usually by assuming a uniform distribution within the buckets. Computing optimal histograms that minimize the error induced by this assumption is NP-hard [25]. The most prominent example of an efficient heuristic is MinSkew [2]. It recursively subdivides the space by splitting the most skewed bucket until the desired number of buckets is reached. Other techniques like rkHist [11] and R-V histogram [1] start from the leaves of a spatial index structure and merge them together for limiting the amount of buckets.

The introduced histograms are, however, not applicable for metric data. There is no straight-forward criterion for subdividing a metric space into a finite amount of disjoint buckets. The missing notion of uniform distribution within a bucket and the unavailability of a volume measure make the incorporation of such buckets into a cardinality estimate impossible. It is possible to transform metric data into vector data in order to build a spatial histogram, though. We can then extract the cardinality estimate for a distance query by calculating the intersection of the query (in form of a hyper-sphere) with the histogram buckets. However, such a transformation into a vector space is costly and introduces an error in form of distance distortions.

Compression techniques like wavelets and cosine transformations are also suitable for cardinality estimation [24]. Both techniques are applicable to multi-dimensional vector data and are shown to provide accurate results. They approximate the actual data distribution by means of a basis function and several coefficients, thus drastically reducing the amount of data. The cardinality estimate is computed as a cumulative joint distribution of the individual dimensions of the data set. However, in a metric space we are not able to use these techniques as the data has no such dimensions and there is no notion of a distribution.

Another method for approximate query processing is Local Sensitive Hashing (LSH). LSH performs very well on data from a high-dimensional vector space. It is for example used for approximate similarity search [15] and thus related to distance queries in metric spaces. There has been work in cardinality estimation of similarity joins using LSH [21]. Also, multiple LSH indexes with different radii can be used for cardinality estimation by counting collisions of hash buckets [14]. However, LSH requires a similarity-preserving hash function which does not universally exist for metric data.

A more recent approach uses Machine Learning [4] for cardinality estimation. It is, to the best of our knowledge, the only method supporting distance queries. The query-driven approach learns to differentiate several prototype queries and predicts the cardinality of unseen queries by assignment to a prototype and subsequent interpolation using regression. The optimization of the query prototypes is performed via gradient descent where the prototype query is moved across the data space. This manipulation of a query object is not possible in a metric space. Thus, like the other approaches, this approach is infeasible for metric data, unless it is mapped into a vector space first.

A distance preserving mapping of data from a metric space to a vector space is called *embedding*. The goal is to find for each $x_i \in X$ an embedding $y_i \in \mathbb{R}^d$, such that the induced *stress* [19] on the distances is minimized. This stress measure incorporates the deviations of the resulting distances among objects with respect to the original distances.

There are different approaches available to embed metric data into a vector space [3]. One prominent example is Multidimensional scaling (MDS) [20]. It tries to preserve the pairwise distances in vector space by using such a stress function [19] and minimizing it subsequently. This minimization can be performed by eigendecomposition or gradient descent. However, both methods are expensive to compute, and thus, not suitable for very large data sets. Landmark MDS [9] was introduced as an alternative to MDS for big data scenarios. It uses samples of the data called landmarks and applies MDS on them. The remaining points are then embedded based on the distances to the l landmark elements.

Kernel estimators [26] are a competitor of histograms which exhibit a fast convergence for 1-dimensional data [7] and have been generalized to multi-dimensional data [16]. Note, that both approaches do not support distance queries on metric data. Here, samples distribute their *weight* using a kernel function K , e.g. Epanechnikov [12] or Gaussian. This weight corresponds to the probability of data points existing in the vicinity of the sample. One approximates the underlying probability density function \hat{f} of a data set at the evaluation point x by using a set of samples S and summing up over all samples: $\hat{f}(x) = \frac{1}{|S| \cdot h} \sum_{s \in S} K(\frac{x-s}{h}) = \frac{1}{|S|} \sum_{s \in S} K_h(x-s)$. Here, h is the smoothing-factor called *bandwidth*. The cardinality estimate results from integrating the kernel density function within a given rectangle query and scaling the result up. In a d -dimensional vector space typically product kernels are used where the density function is integrated for each dimension separately. This is only feasible for rectangular queries and not for distance queries. Hence, the application of existing kernel-density estimators for distance queries on metric data embedded into a vector space is not straight-forward. Approximating the distance query as a hyper-sphere introduces an error that is also influenced by the curse of dimensionality. Our approach makes use of kernels, but we avoid the curse of dimensionality by using the one-dimensional distance function.

The choice of the actual kernel function is considered to be of low impact according to the literature [8]. Nevertheless, we consider different kernel functions in the experiment section of this paper. However, the selection of the kernel bandwidth h has a much more crucial impact on the resulting estimator quality.

There are two general approaches for the bandwidth selection: global and locally adaptive methods [31]. Using a global (fixed) bandwidth means that all samples and evaluation points use the same bandwidth. One method of obtaining this bandwidth is by minimizing the mean integrated squared error (MISE) [30]. In contrast to traditional applications, the underlying distribution that shall be fitted by the kernel estimator is known in cardinality estimation. It is given by the data itself. This enables other optimization techniques than those used in the statistics literature. Recent work [17] used a gradient descent based approach to find the optimal bandwidth for a given set of training queries. They fit a global bandwidth for each dimension of the vector space. However, a global bandwidth is usually not optimal, as the resulting estimator oversmooths the distribution in dense regions and undersmooths in sparse regions of the data set. While the authors of [17] were able to exploit the different distributions in the individual dimensions, we found the error of a global bandwidth for different query sizes in our metric scenario to be significantly high. Furthermore, a gradient descent based approach to bandwidth estimation turned out to get stuck in local optima of poor quality in our experiments. We thus also investigate locally adaptive kernel estimators that vary the bandwidth either based on

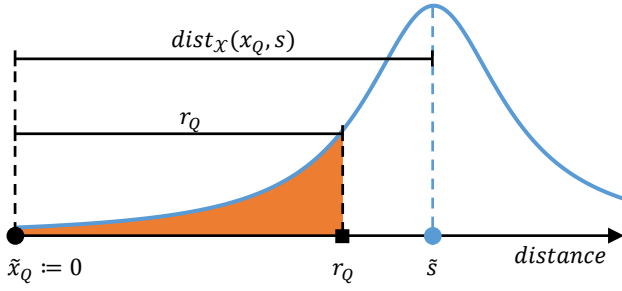


Figure 1: The incorporation of a kernel-sample s into the cardinality estimation for a query $Q = (x_Q, r_Q)$. The omitted y-axis corresponds to the probability density.

Algorithm 1: Generic Kernel Estimation Algorithm

Input : Kernel function $K_h : \mathbb{R} \rightarrow \mathbb{R}_+$, centered at 0
 Optimized bandwidths $\mathcal{B} : X \times \mathcal{X} \times \mathbb{R}_+ \rightarrow \mathbb{R}_+$
 Samples $S \subset X \subseteq \mathcal{X}$
 Total data set size $|X|$
 Distance function $dist_{\mathcal{X}} : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}_+$
 Query $Q = (x_Q, r_Q)$ with object and radius

Output: Estimated cardinality $\tilde{c}(Q)$

```

1 total ← 0.0;
2 foreach s ∈ S do
3   h ← B(s, x_Q, r_Q);
4   s-tilde ← dist_X(x_Q, s);
5   contribution ← ∫_0^{r_Q} K_h(x - s-tilde) dx;
6   total ← total + contribution;
7 end
8 probability ← total/|S|;
9 return ⌊probability · |X|⌋;
```

the sample point or the evaluation point. The latter is also called *balloon estimator* [30].

Other work in kernel-based techniques for cardinality estimation in vector spaces focuses also on improving the efficiency of the estimation process. One approach is reducing the number of samples to a so-called coreset [33] that maximizes both quality and efficiency of the estimator. In the scope of this paper we do not yet consider such improvements but focus on demonstrating the general applicability of kernel estimators to this new scenario of metric data.

4 DISTANCE-BASED KERNEL ESTIMATORS

Kernel estimators allow us to overcome a fundamental problem of using a sample directly for estimating the cardinality of a query result. Namely that the information is concentrated at a sample point. In contrast to a histogram we also get a continuous distribution. In a metric space it is, however, not straight-forward how we can apply a kernel function on a sample point, as there are no dimensions in which they could gradually distribute the mass of a sample. The central idea of our proposed technique is therefore to apply the kernel function on the distance to a sample point in order to incorporate the probability of elements in the vicinity fractionally.

In the following we show how to incorporate a sample point into the cardinality estimate. Here, the query $Q = (x_Q, r_Q)$ with object x_Q and radius r_Q is located at distance $\tilde{s} := dist_{\mathcal{X}}(x_Q, s)$

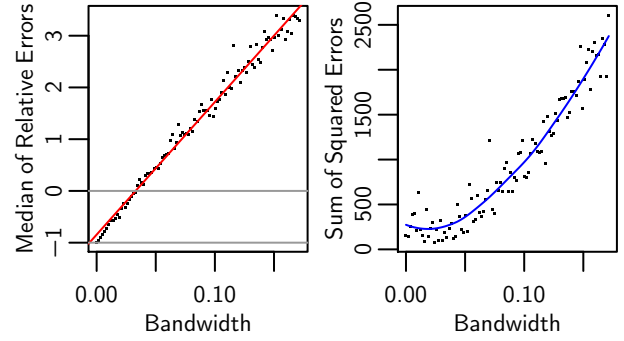


Figure 2: Influence of the bandwidth on the estimation error on the Moby data set (cf. Section 6) for a fixed query size. The left-hand side shows the median of the relative errors (Equation (2)). The right-hand side shows the sum of squared errors (measure M_{LS}).

from the sample point s . As depicted in Figure 1, we introduce an axis expressing the distance to x_Q . For that we map x_Q to $\tilde{x}_Q := 0$, the origin of the axis. The sample point s is then mapped onto \tilde{s} . The kernel function K_h is then centered at point \tilde{s} by subtracting \tilde{s} from its argument. We take the area under the curve of the kernel function between \tilde{x}_Q and r_Q as the *contribution* of this sample to the cardinality estimate.

Algorithm 1 shows the full estimation process. For each sample point we calculate the contribution and compute the sum. For this we first compute the optimized bandwidth by calling the function \mathcal{B} for the given sample point and query with object and radius. In case of a global bandwidth, this function ignores the parameters and always returns the same bandwidth. In case of a locally adaptive approach, it either uses the sample or evaluation point (query) to obtain a specific bandwidth. We detail algorithms for computing the bandwidth in the next section. Given the optimized bandwidth, the distance between sample and query object, and the radius, we calculate the contribution of the sample to the running *total*. After all samples are processed, the probability is then the *total* divided by the number of samples, see line 8. Finally, we scale the resulting probability up by the total data set size and return this value as the cardinality estimate.

The general workflow of our technique consists of (1) collecting a set S of samples, (2) determining the optimal bandwidths \mathcal{B} and (3) applying Algorithm 1 to estimate the cardinality of new queries. In the following we present the process of optimizing the bandwidths.

5 BANDWIDTH OPTIMIZATION

It is well-known [31] that the bandwidth of a kernel function has a crucial impact on the resulting cardinality estimate. A too small bandwidth leads to undersmoothing, a too large bandwidth to oversmoothing. The two edge cases are an infinitely small bandwidth that converges to sampling and an infinitely large bandwidth that converges to a uniform distribution. We thus take particular care of finding an optimal value. We distinguish between a global bandwidth for all samples and queries, and locally adaptive methods where the bandwidth is individually fitted to accommodate for sparser and denser regions of the data space.

5.1 Global

The computation of the optimal global bandwidth for a kernel function and a given data set is an optimization problem. We first formalize this problem and then present our optimization strategy.

5.1.1 Optimization Problem. We want to find a bandwidth h that minimizes the error of estimates for future queries on the given data set. As we do not know the future queries, we extract a set of training queries Q from the data set and minimize the error for these queries. Afterwards, we validate the performance against an independent set of test queries that we extracted from the data set beforehand. We formally define the optimization problem for a fixed kernel function as

$$\arg \min_h \text{Error}_X(h, Q), \quad (1)$$

where h is the bandwidth, X is the data set and Error_X a function that computes the error of the queries Q on X for the given bandwidth h .

We define an appropriate error measure for Equation (1) in two steps. First, we define an auxiliary function

$$\text{error}_X(h, Q) := \frac{\tilde{c}_h(Q) - c(Q)}{c(Q)}, \quad (2)$$

where $\tilde{c}_h(Q)$ is the estimated cardinality using bandwidth h and $c(Q)$ the actual cardinality of query Q on data set X . This measure differs slightly from the common relative error metric, as we do not take the absolute value in the numerator. This allows us to assess over- and underestimates separately. It returns values in the interval $[-1, \infty]$. Two values are of particular interest: -1 indicates that the estimator returns simply a result of zero even though there are results contained in the query. On the other hand, an error of zero indicates a perfect result: the estimated cardinality is equal to the true number of elements the query returns. There is no upper bound for our measure. However, one should notice, that a value of 1 means already an overestimation by a factor of 2.

To compute the error of a set of queries Q we combine the errors $\text{error}_X(Q)$ of the individual queries $Q \in Q$ using a measure $M : \mathbb{R}^{|Q|} \rightarrow \mathbb{R}_+$. M computes for a set of errors E a single value that is then subject to minimization. Two examples for M are the deviation of the median error from zero, and the sum of squared errors (LS for least squares):

$$M_{\text{median}}(E) := | \text{median}(E) |$$

$$M_{\text{LS}}(E) := \sum_{e \in E} e^2.$$

For $M \in \{M_{\text{median}}, M_{\text{LS}}\}$, the final optimization problem is defined as

$$\arg \min_h \text{Error}_X(h, Q) = \arg \min_h M(\{\text{error}_X(h, Q) \mid Q \in Q\}) \quad (3)$$

5.1.2 Optimization Strategy. The minimization of the error function (3) requires an efficient and robust optimization method. Figure 2 shows the relationship between bandwidth and error for an example data set. On the left-hand side of the plot we observe that starting from an infinitely small bandwidth results first underestimate the true cardinality. A higher bandwidth reduces the error to a certain degree. At some point the bandwidth over-smoothes the distribution, leading to very high overestimations. The right-hand side shows the mean squared errors. While the general trend of the error function is clearly visible, we can also

see that the results are noisy. This poses a difficult to find global optimum as the multitude of local optima has to be overcome. A method that has shown to be very effective in practice are Evolution Strategies.

An Evolution Strategy (ES) is a global numeric optimization approach inspired by the Darwinian theory of natural selection. We implemented the approach of Beyer and Schwefel [6]. Here, μ parents produce another set of λ offspring. From the thus obtained set of $\mu + \lambda$ individuals the best μ individuals are selected for the next generation based on a fitness function. An offspring is produced using a recombination of p parents followed by mutation. The evolutionary process is repeated until either a fitness threshold, a number of stall iterations without improvement, or a total number of iterations is reached.

In our case each individual represents a bandwidth h . For calculating the fitness, we use the error function from Equation (3) which we want to minimize. We tried different parametrizations and found the following to provide a good compromise between runtime and estimation errors: $\mu = 100, \lambda = 300, p = 2$, recombination via mean and mutation via a Gaussian distribution. We stop the optimization process after either 25 stall iterations or a given total amount of 1000 iterations is reached.

As the fitness function has no side-effects and only depends on the parameter h we can easily parallelize the evolutionary process by computing the score for different individuals on different threads. Because the evaluation of the fitness function is the most expensive part of the computation we can effectively scale up the throughput linearly in the amount of CPU cores.

We also exploit the fact that only a limited number of distinct distance computations are required. In the fitness evaluation for a given bandwidth h we need to compute a new estimate for each training query based on h . However, the distances of the query objects to the samples that give the interval for the integral in line 5 of Algorithm 1 remain constant. In our implementation we thus precompute these distances and store them in a matrix \mathcal{D} . In the iterations of the optimization process we then avoid redundant recomputations by performing simple look-up operations. The matrix \mathcal{D} is of size $|S| \cdot |O|$, where S is the set of samples and $O := \{x_Q \mid (x_Q, r_Q) \in Q\}$ is the set of distinct training query objects. \mathcal{D} is discarded after the optimization process is finished. It thus only influences the construction-time space complexity of our algorithm. Reasonable numbers are 10^2 distinct query objects and 10^4 samples as used in our experiments. This means \mathcal{D} typically requires only a few megabytes of memory.

5.2 Locally Adaptive

A fixed global bandwidth is usually insufficient for approximating the cardinality of small and large queries in dense as well as sparse regions. One possibility is to extend the bandwidth optimization process from the previous subsection to individual bandwidths per sample point. For the following reasons we refrain from this direction and rather focus on the balloon estimator [30].

In the Evolution Strategy we can extend the configuration to incorporate individual bandwidths. However, this leads to a very high-dimensional optimization problem as we potentially need to optimize hundreds or thousands of bandwidths simultaneously. Such an optimization may take very long to converge or get stuck in a local optimum of overall poor quality.

Another approach is to use coordinate search, where we optimize only one bandwidth at a time, keeping the others fixed. Here, we would initialize the N individual bandwidths h_i e.g.

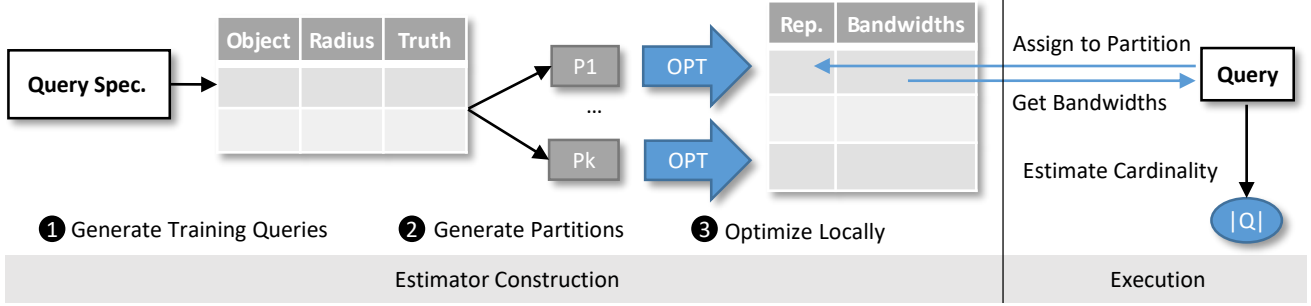


Figure 3: A general overview of the locally adaptive estimator construction and execution.

randomly and then optimize them one at a time in the order $i = 1, \dots, N, N - 1, \dots, 1$. We repeat this until no significant improvement is made in one full sweep. This approach, however, is also prohibitively expensive to compute.

We thus limit our investigation to the balloon estimator defined by the k^{th} -nearest neighbor estimator [23]. It chooses the bandwidth on basis of the evaluation point (query) and the surrounding sample points. In contrast to the traditional balloon estimator, we again use a set of training queries in the optimization process. Given a query $Q = (x_Q, r_Q)$ and the samples S we choose the bandwidth h as follows:

$$h = \mathcal{B}(\cdot, x_Q, r_Q) \propto \text{distance}(x_Q, \text{sample}_k(x_Q)), \quad (4)$$

where $\text{sample}_k(x_Q) \in S$ is the k^{th} -nearest neighbor of the query object x_Q in S , and \mathcal{B} the function returning the optimized bandwidth as introduced in Algorithm 1. We thus ignore the first parameter of \mathcal{B} and apply the same bandwidth to all samples in the final cardinality estimation of the query Q . Obviously, the distance to the k^{th} -nearest neighbor is smaller in denser regions than it is in sparser regions. This leads to the desired adaptive bandwidth.

Using the distance to the k^{th} -nearest neighbor directly, however, turns out to be insufficient. First of all, we need a linear scaling factor to transform the distance to a bandwidth. Furthermore, we incorporate the query radius into the bandwidth optimization as well. As our experiments will show, the optimal bandwidth varies for different query sizes (selectivity). However, we only know the radius at query time. The result size is precisely the value we want to estimate.

We can, however, distinguish differently sized queries by the relationship between the query's radius and the distance to the k^{th} -nearest neighbor sample. Here, the latter is an indicator of the density around the query object. We denote this relationship as

$$\rho(Q) = \frac{r_Q}{\text{dist}_X(x_Q, \text{sample}_k(Q))}. \quad (5)$$

It is an indicator for the cardinality of the query. If we fix the query radius, $\rho(Q)$ gets larger the smaller the distance to the k^{th} -nearest neighbor gets. This matches our intuition as this indicates the data space becoming more dense. On the other hand, if we fix the density in form of the denominator, a larger radius corresponds to a larger ρ value, indicating a larger cardinality.

For our extended balloon estimator called *BalloonEstimator⁺* (B^+ for short) we thus optimize the local bandwidth for a given query Q depending on $\rho(Q)$. Here, we again make use of the methods presented in the previous subsection. The optimization process consists of the following steps which are also presented in Figure 3:

1. Generate a set Q of training queries based on a query specification (e.g. m query objects with a set of target selectivities). This involves calculating for a given query object x_Q a radius r_Q and the true cardinality that fulfill the query specification.
2. Divide Q into k disjoint partitions $P_i \subseteq Q$ such that the intra-partition variance $\text{Var}(\{\rho(Q) \mid Q \in P_i\})$ is minimized.
3. For each partition P_i compute the optimal bandwidth h_{P_i} that minimizes $\text{Error}_X(h_{P_i}, P_i)$.

At query time we assign the query to a partition and use the optimized bandwidth.

In order to obtain the k partitions required in Step 2 we use *hierarchical* clustering with complete linkage based on ρ . We extract the k partitions from the dendrogram. For each partition we then compute the locally optimal bandwidth by means of an ES. Each partition is then identified by the mean ρ -value over all queries in the partition: $\bar{\rho}(P_i) = \frac{1}{|P_i|} \sum_{Q \in P_i} \rho(Q)$. Given a new query Q we first assign it to the nearest partition P^* by

$$P^* = \arg \min_{P_i} |\bar{\rho}(P_i) - \rho(Q)|.$$

We then use h_{P^*} as the bandwidth for the kernel functions when estimating the cardinality of query Q .

The depicted construction process of the B^+ estimator requires several distance computations and incurs some overhead in storage. For calculating the k -nearest neighbor when determining the ρ -value of the training queries we use the same matrix \mathcal{D} as for the actual bandwidth optimization. Storing the B^+ estimator requires storing for each cluster the mean ρ -value and the optimized bandwidth. This slightly reduces the amount of samples that can be stored with a given amount of space.

When estimating the cardinality for a given query Q , we induce some overhead with respect to a kernel estimator with a global bandwidth. Namely, we need to compute $\rho(Q)$ which includes finding the k^{th} -nearest neighbor sample. However, we calculate the distances of all samples to the query object anyway when calculating the estimation after the bandwidth has been determined. Thus, the additional overhead of the B^+ estimator only consists of finding the k -smallest element in the list of distances. We can efficiently determine this element while computing the distances using a bounded max-heap. Here, we always store the k smallest elements and larger elements are discarded. At the end the k -smallest distance is at the top of the heap. We then incorporate this distance in the calculation of $\rho(Q)$ and retrieve the bandwidth of the nearest partition. The remaining computations are identical to a kernel estimator with a global bandwidth.

6 EXPERIMENTAL EVALUATION

This section presents the results of our experimental study. First, we present the experiment setting, data sets and investigated methods. Then, we present the results of the experiments.

6.1 Setting

We implemented the kernel estimators and baseline algorithms in Java using embedding techniques and histograms from the XXL library [10]. All experiments were run on an Intel Core i7-4771 CPU and 16GB of RAM. We consider four real-world data sets which, together with our generated queries, are on our website¹.

Moby Word List (Moby) contains word lists in different languages². We used the German word list consisting of about 160 000 words and the Levenshtein distance [22] as a metric.

Protein Binding Sites (PBS) are taken from CavBase [29], a database of protein binding sites (PBS) from experimentally determined protein structures. In CavBase, currently 248 686 PBS are stored. Each PBS is described by a set of points in the 3-dimensional Euclidean space that model the shape of the protein. In addition to the coordinate, each point also carries a label specifying the physico-chemical property of that point. To compare pairs of PBS we use the measure presented in [13] which fulfills all metric properties.

Rea16 is a 16-dimensional vector-data set representing Fourier coefficients from CAD data. It contains 1.3 million points and is often used for benchmarking high-dimensional queries [5]. We use the Euclidean distance measure as metric.

Wikipedia (Wiki) is a data set consisting of 4.5 million Wikipedia articles represented by the article name and a short abstract. We used the data provided by the DBpedia project³. We removed articles that are only a disambiguation or a list of links to other articles. For easier syntactic comparisons we also removed all non-ASCII characters. The distance measure for this data set is based on the Jaccard coefficient of character shingles⁴ of length $k = 3$. The distance between two articles is computed by $dist(a, b) = 1 - \frac{|s_k(a) \cap s_k(b)|}{|s_k(a) \cup s_k(b)|}$, where a and b are articles and s is a function that computes the set of k -shingles of a short abstract.

We generate our test queries uniformly at random from the data set at hand. For this, we select 100 items as query objects and remove them from the data set. In particular we remove these items from the data set before optimizing the kernel bandwidths on a different set of training queries. We then choose the minimum radius around these items such that at least 0.01%, 0.1% and 1% of the data set are contained in the query. This radius varies depending on the density at the particular query object. The training queries are generated in the same fashion as the test queries. Note that it is not always possible to create a query from a data object that contains exactly the desired amount of data. Especially for discrete distance measures, many objects share the same distance to a reference object. Also note that by generating queries this way, they follow the distribution of the data. This means that it is more likely for a query to be placed in a more dense region of the data space. However, this is no undesired effect, since this approach intuitively matches real-world query patterns [17]. In contrast to vector spaces it is not possible to uniformly sample queries from a domain for metric data (e.g. of

newspaper articles). We are thus forced to limit our investigation to queries drawn uniformly from the data items.

In our experiments we compare the estimated cardinality for a given query with the actual cardinality – the closer both values, the better the estimator. We use our error measure from Equation (2) again. Recall that we use this slight variation of the common relative error to capture under- and overestimations individually.

For measuring the build time of the individual estimators we use wall-clock time. Note that we parallelized the Evolution Strategy and used all 8 threads of the machine. For our kernel-based approaches we measure only the time for creating the estimator and not for generating the training queries as, in practice, they can e.g. be obtained from historic queries or the current workload.

In addition to our kernel estimators we consider spatial histograms as a baseline method for cardinality estimation. For metric data, the spatial histograms require an embedding into a vector space. We use the following popular measure to assess the quality of the resulting embeddings:

$$STRESS_1 = \left(\frac{\sum_{i < j} (dist_{\chi}(x_i, x_j) - dist_E(y_i, y_j))^2}{\sum_{i < j} dist_{\chi}(x_i, x_j)^2} \right)^{\frac{1}{2}}, \quad (6)$$

where $dist_E$ is the Euclidean distance. It measures on a relative scale how well distances are preserved by the specific embedding. According to the literature [19] a value greater than 0.2 is already considered as a low-quality embedding.

We evaluated the performance of different kernels with optimized bandwidths. Even though the concrete choice of kernel function is considered to be insignificant in the literature [8], we found differences among the optimization results. We consider Cauchy, Epanechnikov, Exponential and Gauss kernels.

We now describe the estimators we used in our experiments. All of them are given 0.1% of the total data set size in memory to allow a fair comparison across the data sets of varying sizes. This size presents a reasonable value in big data scenarios and returned representative results in our experiments.

Sampling uses reservoir sampling to efficiently build a uniform random sample S of the data in one pass. The query is executed on the sample and the resulting cardinality $c_S(Q)$ is scaled up to return the estimated cardinality $\tilde{c}(Q) = c_S(Q) \cdot |X|/|S|$.

MSnd and R-Vnd use a MinSkew or R-V histogram, respectively. The parameter n indicates that the data was first embedded in an n -dimensional vector space using Landmark MDS. A distance query Q is then also embedded and becomes an n -dimensional query sphere \hat{Q} . For estimating the cardinality we calculate the volume $V_I(\hat{Q}, B)$ of the intersection between \hat{Q} and histogram bucket B and sum up the fractional bucket counts:

$$\tilde{c}(Q) = \sum_B V_I(\hat{Q}, B) \cdot count(B).$$

For $n > 2$ we use a Monte Carlo simulation for V_I , as it is more efficient than an analytical solution.

Global size uses a kernel estimator with the Cauchy kernel function and a global bandwidth optimized by means of an Evolution Strategy, as described in Subsection 5.1, with 100 individuals. We use M_{median} as defined in Section 5. The parameter *size* gives the cardinality of the training queries: $S \hat{=} small \hat{=} 0.01\%$, $M \hat{=} medium \hat{=} 0.1\%$, $L \hat{=} large \hat{=} 1\%$, $MIX \hat{=} X \hat{=} mixed \hat{=} SUMUL$. The query objects are selected uniformly at random.

B⁺k,n uses a kernel estimator with the Cauchy kernel and locally optimized bandwidths as described in Subsection 5.2 using

¹<http://uni-marburg.de/qiFqp>

²<http://icon.shef.ac.uk/Moby>

³<http://wiki.dbpedia.org/services-resources/documentation/datasets#ShortAbstracts>

⁴A k -shingle is a sequence of k characters

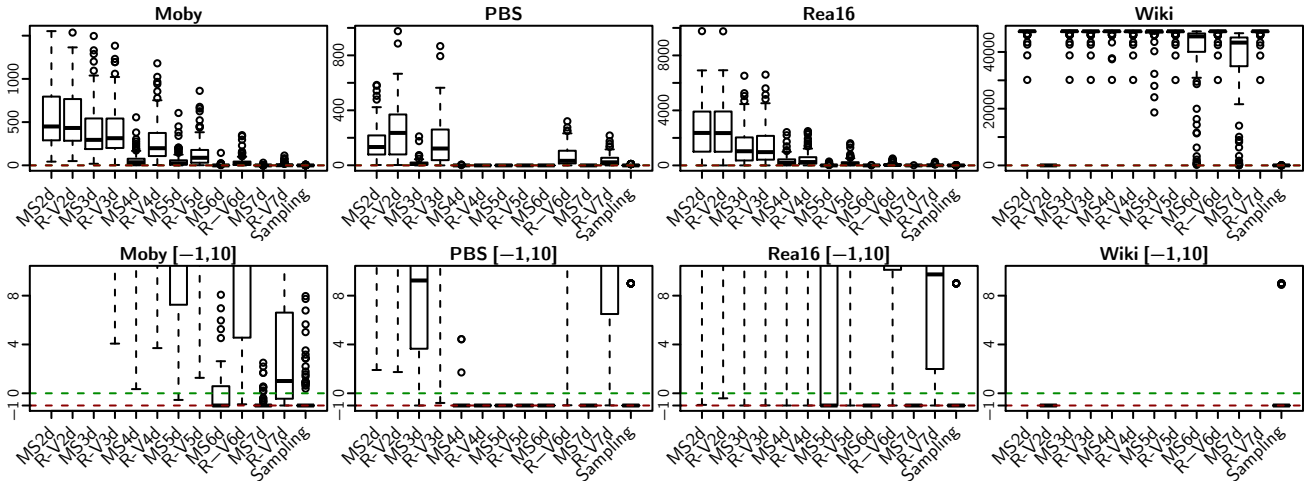


Figure 4: Estimation errors of the spatial histograms on queries of 1% of the data. The green line indicates an error of 0, the red line an error of -1 (zero-estimate). The top row plots show the total error range, the bottom row the interval $[-1, 10]$.

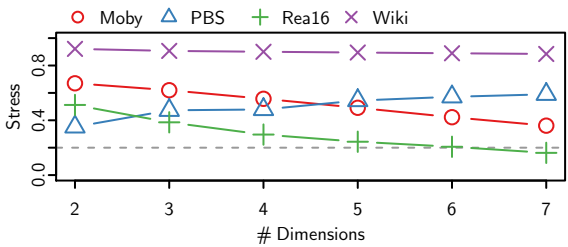


Figure 5: The stress induced by the embedding into a vector space of d dimensions. The dashed line indicates the threshold for a low-quality embedding.

the same measure as *Global* on the *MIX* training queries. The parameter k expresses that the k^{th} -nearest neighbor is used and n defines the number of clusters.

6.2 Estimation Quality

We now present the results of our experiments of the described estimators on the introduced data sets. We visualize the estimation errors in terms of boxplots.

6.2.1 Baseline Algorithms. Figure 4 shows the estimation errors of MinSkew and R-V histogram for the individual data sets and a query size of 1%. We embedded all data sets into 2, 3, 4, 5, 6 and 7 dimensions using Landmark MDS with 10 landmark points. We can immediately see very large estimation errors. Using a higher target dimension for the embedding leads to lower estimations. However, this does not lead to good estimates. In contrast, the very high estimates turn into zero-estimates ($error = -1$) for higher dimensions.

Figure 5 shows the stress induced by the embedding. Overall, the embeddings are of poor quality with stress values above 0.2 as indicated by the dashed line. We observe that, in general, the stress decreases with the number of used dimensions. For PBS we notice a slightly worse embedding for higher dimension. This reveals a weakness of the Landmark MDS embedding. Only the landmarks profit directly from the higher degree of freedom in a higher dimensional vector space. They are properly embedded using an MDS that minimizes the stress. The remaining elements

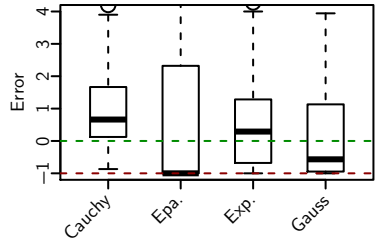


Figure 6: Influence of the choice of kernel function on the Moby data set for a query size of 0.01%.

are only embedded with respect to the landmarks, not with respect to the other objects. Thus, a higher dimensionality does not necessarily guarantee a better embedding.

In addition to the poor embedding quality, the spatial histograms suffer from the curse of dimensionality. This leads to an overall performance degradation with an increasing dimensionality. The vector space becomes sparse leading to almost empty buckets or queries that intersect with no buckets at all. This is apparent as the high overestimations induced by the embedding turn into zero-estimates for higher dimensions. We conclude that spatial histograms are not a reliable choice for cardinality estimation in metric spaces. For the vectorial Rea16 data set in the original 16-dimensional vector space (omitted in Figure 4 due to lack of space), nearly all queries return an estimate of zero. This motivates the idea of using distance-based approaches on high-dimensional data in order to counter the curse of dimensionality.

Figure 4 also shows the performance using a random sample for estimating the cardinality. The resulting cardinality on the sample is scaled up by the constant factor $|X|/|S|$, where S is the sample and X the data set. If the sample is large enough with respect to the query size it reflects the data distribution well and estimates become accurate. However, if the sample is small or the query highly selective, errors increase dramatically. Estimates are either zero because no sample is contained in the query, even though there are qualifying items in the underlying data set. Or, estimates are very high because by chance a sample was included and then scaled up by a large factor. This results in a median of -1 and a very high maximum error.

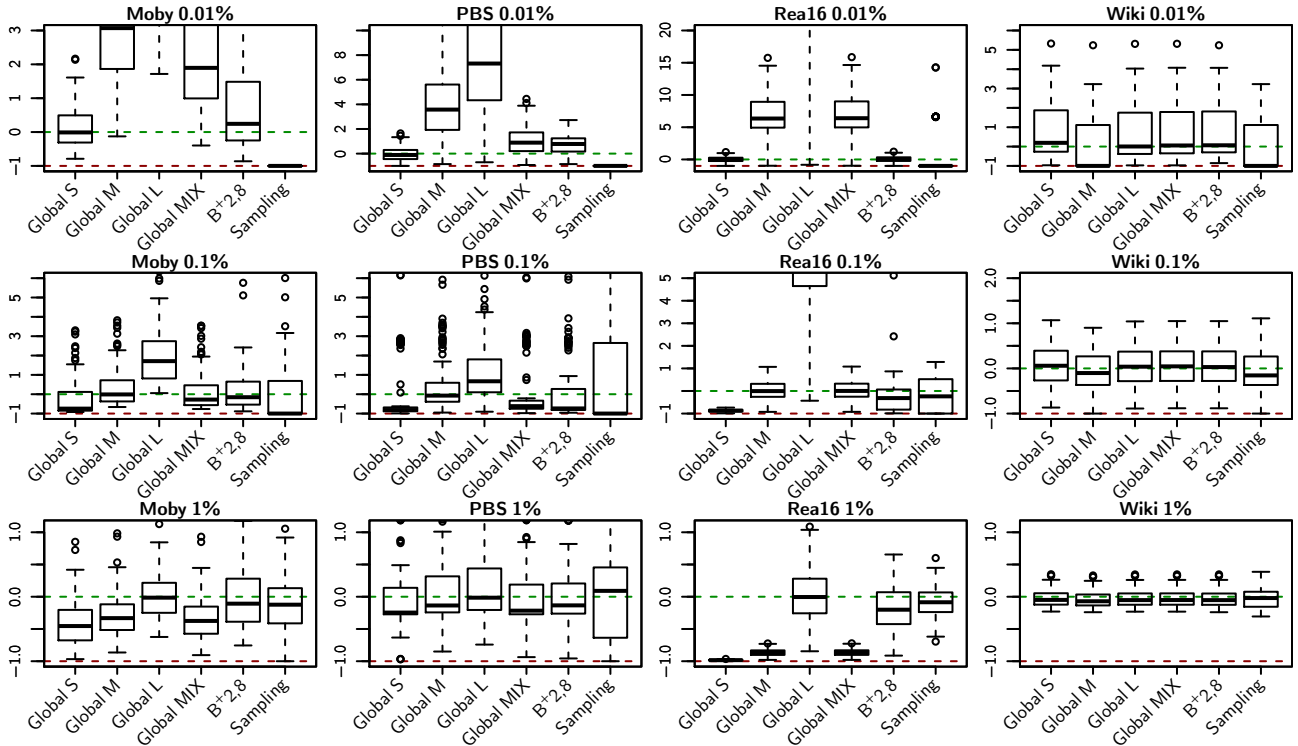


Figure 7: Estimation errors of the kernel estimators with global and local optimized bandwidths for different query sizes.

6.2.2 Kernel-Based Techniques. Figure 6 exemplifies the performance of different kernel functions using an optimized bandwidth as described in Section 5. It is apparent that different kernel functions lead to a different estimation accuracy. However, our experiments confirm that the selection of the bandwidth has a much greater impact on the performance than the choice of the kernel function. As the Cauchy kernel gave the overall best performance in our experiments, we limit our following presentation to this kernel. The reason for its superior performance is that its distribution is more heavy-tailed than that of the other kernels. In contrast to, e.g., the Gaussian kernel, the density does not decrease exponentially in the distance from the mean. This allows the kernel to distribute its weight more effectively.

Figure 7 shows the estimation errors of the kernel estimators with globally and locally optimized bandwidths. We also report the statistics in Tables 2-5 at the end of the paper. In contrast to sampling the kernel estimators are able to give reasonable cardinality estimations also for very selective queries. In particular they overcome the problem of either zero estimates or very high overestimates. For the global bandwidth optimization we generated training queries of different cardinality. In Figure 7 the labels *S*, *M*, *L* and *MIX* indicate the target query size for the bandwidth optimization process.

Kernel estimators optimized for a specific query size give good results on new queries of equal size. This proves that the bandwidth generalizes well from training to test queries. However, estimators optimized for small queries tend to underestimate the cardinality of larger queries. The optimal bandwidth for larger query sizes also leads to overestimates for smaller queries. Optimizing the bandwidth for all query sizes at the same time leads to a bandwidth that is not optimal for any of the query sizes. This result shows that a global bandwidth is not sufficient for applications where queries vary greatly in size.

The *BalloonEstimator*⁺ (B^+k, n) uses the same set of training queries, but computes a set of optimal bandwidths. The concrete bandwidth for a given query is chosen based on the local density and the query radius. We restrict our results to the setting $k = 2$ and $n = 8$ that gave slightly better results than for other settings.

Figure 8 displays how many results of good quality are produced by sampling and $B^+2, 8$, respectively. It confirms the results indicated by the boxplots that the kernel approach outperforms sampling for small queries. The plot shows the percentage of estimates within the error bounds given by the x-axis. Thus, it begins with all estimates with error zero, meaning perfect results. It then gradually increases the bounds uniformly in both directions, thus accumulating more results. The x-axis ends with all queries of errors in the interval $]-1, 1[$, as we consider greater errors as a failure of the estimator. Accordingly, we can see the number of estimates not fulfilling our requirements by the height of the plot. For all data sets the smallest queries show no results for sampling with absolute errors below 1. $B^+2, 8$ on the other hand is able to produce high quality results within the shown interval. Moreover, the shape of the curve is very consistent between the different query sizes and data sets, indicating the robustness of the approach. Sampling shows step-wise increments when new sample points are incorporated into the estimate.

The concrete choice of n and k is subject to optimization. We report one setting ($k = 2, n = 8$) that worked for all our data sets. There is a general tradeoff between the number of clusters (improving the estimation quality) and the required space (worsening the quality). In general, we observed in our experiments that large values for n and k do not improve the performance. Overall we conclude that the *BalloonEstimator*⁺ presents a significant improvement over the global bandwidth approach.

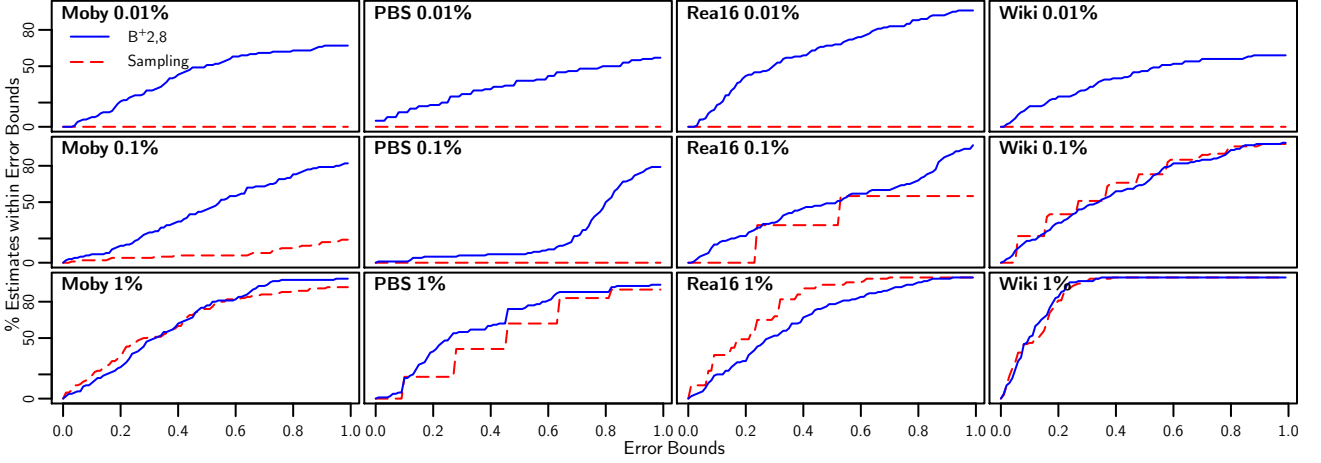


Figure 8: The plotted lines display the number of estimates with absolute errors within the bounds, given by the x-axis.

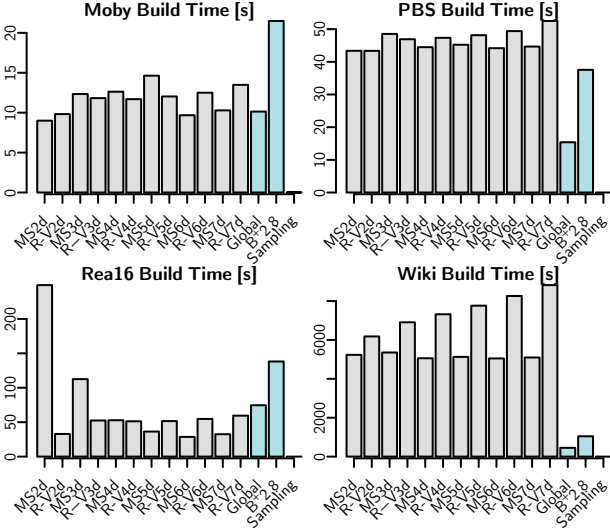


Figure 9: The build time of the estimators in seconds.

6.3 Runtimes

The efficiency of an estimator is the second criterion for assessing its performance. We first discuss the theoretical complexity of the construction of our B^+ estimator. Then, we present our empirical findings of the build and query time of all investigated methods.

6.3.1 Construction Complexity of the B^+ Estimator. The time complexity of the construction of our B^+ estimator is determined by (1) the cost C_P for creating P partitions of the training queries and (2) the cost C_{ES} of executing the ES which is dominated by evaluating the fitness of the individuals. All other operations are negligible because they cause only constant overhead.

The number of partitions P is defined by the user. The cost for partitioning the training queries Q using hierarchical clustering is dominated by the construction of the dendrogram which requires $C_P = O(|Q|^2)$ time.

The fitness evaluation of a single individual requires calculating the cardinality estimate for each training query in Q by incorporating the contribution of all samples S . We denote this cost as $c_{eval} = O(|Q| \cdot |S|)$. Given the number of iterations I , the number of parents μ and the number of offspring λ the cost for

executing the ES is $C_{ES} = O((\mu + I \cdot \lambda) \cdot c_{eval})$. As we execute one ES for each of the P partitions, the final time complexity is $O(|Q|^2 + P \cdot (\mu + I \cdot \lambda) \cdot |Q| \cdot |S|)$.

The time complexity depends on the number of iterations I of the ES. The termination condition is a given number of stall iterations \bar{I} and thus depends on the fitness development of the individuals. We can consider I as a random variable. For $\bar{I} = 25$, we observed the following number of iterations in our experiments: min. 79, 1st quantile 86, median 89, mean 93.57, 3rd quantile 101 and max. 136. Note that we also limit the ES to at most 1 000 iterations which gives an upper bound on the runtime.

6.3.2 Empirical Findings. Figure 9 shows the build times of the estimators from Subsection 6.2. The spatial histograms show a more or less constant construction time, regardless of the dimension. Sampling is the cheapest estimator to construct as it only requires a single scan of the data set. In comparison, our kernel-based techniques take more time to build. As our B^+ estimator performs multiple bandwidth optimization for the different query partitions, its runtime exceeds the global bandwidth approach. However, the required time for construction is still very reasonable in practical applications, especially as the estimator has to be built only once which makes the query response time much more crucial. We observed a very quick convergence of the Evolution Strategy in our experiments, where an optimum was usually found within very few iterations. We report the average build time for the different used configurations. The bandwidth optimization was performed in parallel, while the spatial histograms were constructed on a single thread. As the ES is very easy to parallelize, we can thus improve the construction time using more CPU cores. We also want to emphasize that a stochastic approach is only one method to bandwidth optimization. If another method turns out to be more efficient in this regard, it does not lower the accuracy of a kernel estimator. However, the build time is already very reasonable for practical scenarios. For Moby and PBS the construction of our estimator took less than a minute and for Rea only slightly more than two minutes. This means the overhead of constructing the estimator while importing the data into a database is insignificant. On the Wiki data set the construction was significantly slower with ~17 minutes. This has two reasons: The first is that the estimator size is much larger, as we choose it in relation to the data set size in our experiment. The second reason is that the distance computations

Table 1: Mean query times of the estimators in ms.

Method	Moby	PBS	Rea16	Wiki
MS2d	0.232	0.210	0.985	1.825
R-V2d	0.075	0.935	1.229	0.752
MS3d	5.114	114.848	438.672	363.161
R-V3d	4.968	723.424	438.997	4 868.446
MS4d	4.391	41.765	158.357	97.783
R-V4d	4.270	0.055	384.439	4 239.452
MS5d	4.072	21.134	35.995	35.169
R-V5d	4.020	0.051	351.146	3 854.658
MSd	0.203	0.227	0.203	0.967
R-V6d	3.551	915.755	328.374	3 609.655
MS7d	0.224	0.278	0.231	0.992
R-V7d	3.480	872.779	313.527	3 429.448
Kernel Global	0.105	1.032	0.209	354.941
B ⁺ 2,8	0.138	1.071	0.299	348.178
Sampling	0.104	1.412	0.241	351.645

are the most complex of all data sets. This is also visible in the high construction times of the spatial histograms.

Table 1 shows the query response times of the cardinality estimation. The spatial histograms suffer from slow response times for dimensions greater than 2 which is amplified by the more complex calculation of the sphere-bucket intersection. For some experiments the time gets close to zero as no intersecting buckets were found due to the exponentially increasing vector space. Sampling, again, has the benefit of being very fast, as only few distance calculations are necessary. The kernel estimators additionally need to compute an integral for every sample point which results in a longer execution time. However, as reflected by the query times, this turns out to be insignificant overhead in comparison to the distance computations. The B⁺ estimator is also very efficient. Our experiments confirm that the determination of the appropriate bandwidth induces only very little overhead. In conclusion kernel-based cardinality estimation is an efficient method with effectively zero overhead in comparison to sampling in practical applications. The only reason to use sampling is its low cost for computing the estimator.

6.4 Impact of Estimator Size

The estimator size limits the number of samples used for calculating the cardinality estimate and thus impacts the quality of the estimates. We expect that a larger estimator size leads to overall better estimates. For estimator sizes of 0.05%, 0.1%, 0.2% and 0.4% of the total data set size we measured the performance of the estimators in 10 runs using different samples. In each run we, again, measure the error of 100 test queries. We compute the trimmed mean of the thus generated 100 error values with a ratio of 10% to accommodate for outliers. Figure 10 shows boxplots of the trimmed means for our B⁺2,8 estimator and pure sampling with a query selectivity of 0.01%. For both estimators we observe an overall improvement of the estimates with greater estimator sizes. Our B⁺2,8 estimator is able to already give good results when given only a very limited amount of space. The improvements for larger estimator sizes are relatively small for all data sets. In contrast, we can see that sampling benefits much greater from more available data. It starts with very bad results for such selective queries with a median of zero when only few sample are used. Overall we observe a tendency to underestimate while

Table 2: Estimation errors on the Moby data set.

	Glob S	Glob M	Glob L	Glob X	B ⁺ 2,8	Sampling	
0.01%	Min	-0.790	-0.128	1.717	-0.397	-0.863	-1.000
	Median	-0.011	3.070	11.594	1.896	0.243	-1.000
	Max	7.613	12.167	40.111	8.581	57.842	61.750
0.1%	Min	-0.919	-0.661	0.056	-0.765	-0.884	-1.000
	Median	-0.744	-0.011	1.711	-0.277	-0.155	-1.000
	Max	3.302	3.823	6.348	3.546	6.869	6.007
1%	Min	-0.967	-0.864	-0.624	-0.906	-0.755	-1.000
	Median	-0.455	-0.331	-0.014	-0.375	-0.107	-0.123
	Max	0.852	0.984	1.505	0.931	1.177	1.703

Table 3: Estimation errors on the PBS data set.

	Glob S	Glob M	Glob L	Glob X	B ⁺ 2,8	Sampling	
0.01%	Min	-1.000	-0.852	-0.704	-0.926	-0.852	-1.000
	Median	-0.111	3.574	7.315	0.889	0.782	-1.000
	Max	37.111	41.667	46.333	38.333	37.593	36.037
0.1%	Min	-0.993	-0.949	-0.905	-0.978	-0.956	-1.000
	Median	-0.818	-0.071	0.670	-0.617	-0.735	-1.000
	Max	6.442	7.011	7.551	6.595	7.022	6.299
1%	Min	-0.969	-0.850	-0.742	-0.936	-0.957	-1.000
	Median	-0.246	-0.136	-0.015	-0.216	-0.134	0.091
	Max	1.538	1.507	1.485	1.529	1.546	2.638

the errors of our approach are more centered around zero. With greater estimator size, sampling is able to catch up with our kernel-based approach for some data sets. However, small estimator sizes with respect to the total data set size are highly relevant in practical scenarios because the amount of data in databases is rapidly growing. This makes our estimator superior to sampling in practical scenarios.

7 CONCLUSION AND FUTURE WORK

We presented the first effective and efficient approach to cardinality estimation on metric data. Our approach is based on the application of kernel-density techniques to the one-dimensional distance function. We are able to outperform sampling which suffers from zero-estimates and large overestimates for small queries. Our approach is also superior to spatial histograms on embedded data which suffer both from the poor quality of embeddings and the curse of dimensionality. We presented approaches for determining the bandwidth of the kernel estimator globally and locally adaptive.

In our future work we will further investigate efficient local bandwidth optimization strategies. Furthermore, we will look into adapting our estimator to changing data and query workloads by performing the ES continuously in the background. We will also address the incorporation of query feedback into the estimation process.

REFERENCES

- [1] Daniar Achakeev and Bernhard Seeger. 2012. A Class of R-Tree Histograms for Spatial Databases. In *SIGSPATIAL/GIS*. ACM, New York, NY, USA, 450–453.
- [2] S. Acharya, V. Poosala, and S. Ramaswamy. 1999. Selectivity estimation in spatial databases. *ACM SIGMOD Record* 28, 2 (1999), 13–24.
- [3] Charu C Aggarwal. 2015. *Data mining: the textbook*. Springer.
- [4] Christos Anagnostopoulos and Peter Triantafyllou. 2017. Query-Driven Learning for Predictive Analytics of Data Subspace Cardinality. *ACM Trans. Knowl.*

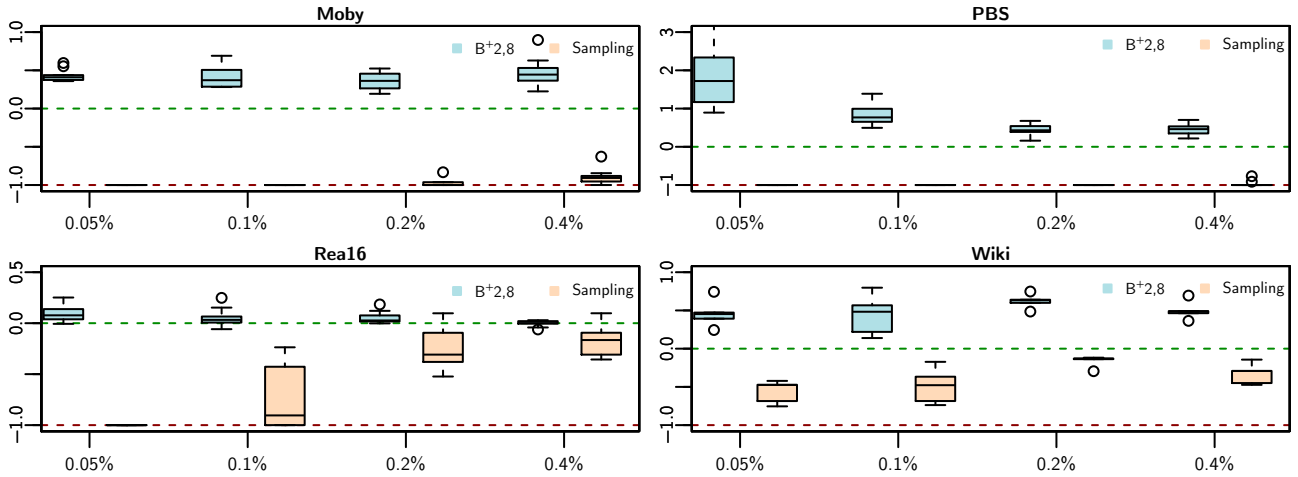


Figure 10: The trimmed mean estimation errors for estimators of varying sizes on a fixed set of 100 queries with target selectivity of 0.01%. The estimator sizes are given in % of the total data set size. For each size, the boxplot includes the results of 10 runs with different samples.

Table 4: Estimation errors on the Rea16 data set.

	Glob S	Glob M	Glob L	Glob X	B ⁺ 2,8	Sampling
0.01%	Min	-1.000	-0.977	-0.826	-0.977	-1.000
	Median	-0.061	6.347	54.966	6.397	-0.004
	Max	1.145	15.756	126.191	15.870	1.206
0.1%	Min	-0.991	-0.925	-0.431	-0.925	-0.990
	Median	-0.873	-0.004	6.583	0.003	-0.314
	Max	-0.736	1.069	14.708	1.082	5.122
1%	Min	-0.997	-0.980	-0.845	-0.979	-0.913
	Median	-0.983	-0.869	-0.004	-0.868	-0.201
	Max	-0.965	-0.725	1.087	-0.723	0.655

Table 5: Estimation errors on the Wiki data set.

	Glob S	Glob M	Glob L	Glob X	B ⁺ 2,8	Sampling
0.01%	Min	-0.966	-1.000	-0.973	-0.971	-0.856
	Median	0.192	-1.000	0.004	0.057	0.065
	Max	5.324	5.237	5.310	5.314	5.237
0.1%	Min	-0.867	-1.000	-0.888	-0.882	-0.883
	Median	0.060	-0.102	0.039	0.046	0.030
	Max	1.068	0.901	1.042	1.049	1.048
1%	Min	-0.230	-0.240	-0.231	-0.231	-0.240
	Median	-0.049	-0.070	-0.052	-0.051	-0.052
	Max	0.349	0.330	0.346	0.347	0.346

Discov. Data 11, 4, Article 47 (June 2017), 46 pages.

- [5] Norbert Beckmann and Bernhard Seeger. 2009. A Revised R⁺-tree in Comparison with Related Index Structures. In *SIGMOD Conf.* ACM, NY, USA, 799–812.
- [6] Hans-Georg Beyer and Hans-Paul Schwefel. 2002. Evolution strategies – A comprehensive introduction. *Natural Computing* 1, 1 (2002), 3–52.
- [7] Björn Blohfeld, Dieter Korus, and Bernhard Seeger. 1999. A Comparison of Selectivity Estimators for Range Queries on Metric Attributes. In *SIGMOD Conference*. ACM Press, New York, NY, USA, 239–250.
- [8] Noel Cressie. 2015. *Statistics for spatial data*. John Wiley & Sons.
- [9] V. de Silva and J. B. Tenenbaum. 2004. *Sparse multidimensional scaling using landmark points*. Technical Report. Stanford University.
- [10] Jochen Van den Bercken, Björn Blohfeld, Jens-Peter Dittrich, Jürgen Krämer, Tobias Schäfer, Martin Schneider, and Bernhard Seeger. 2001. XXL – A Library Approach to Supporting Efficient Implementations of Advanced Database Queries. In *VLDB*. Morgan Kaufmann, San Francisco, CA, USA, 39–48.
- [11] Todd Eavis and Alex Lopez. 2007. Rk-hist: An R-tree Based Histogram for Multi-dimensional Selectivity Estimation. In *CIKM*. ACM, NY, USA, 475–484.
- [12] V. A. Epanechnikov. 1969. Non-Parametric Estimation of a Multivariate Probability Density. *Theory of Probability & Its Applications* 14, 1 (1969), 153–158.
- [13] Thomas Fober, Marco Memberger, Gerhard Klebe, and Eyke Hüllermeier. 2010. Efficient Similarity Retrieval of Protein Binding Sites based on Histogram Comparison. In *GCB (LNI)*, Vol. 173. GI, Bonn, Germany, 51–59.
- [14] Jinyang Gao, H. V. Jagadish, Beng Chin Ooi, and Sheng Wang. 2015. Selective Hashing: Closing the Gap between Radius Search and k-NN Search. In *KDD*. ACM, New York, NY, USA, 349–358.
- [15] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. 1999. Similarity Search in High Dimensions via Hashing. In *Proc. of the 25th Int. Conf. on Very Large Data Bases (VLDB '99)*. Morgan Kaufmann Publishers Inc., SF, CA, USA, 518–529.
- [16] Dimitrios Gunopulos, George Kollios, Vassilis J. Tsotras, and Carlotta Domeniconi. 2000. Approximating Multi-Dimensional Aggregate Range Queries over Real Attributes. In *SIGMOD Conference*. ACM, New York, NY, USA, 463–474.
- [17] Max Heimerl, Martin Kiefer, and Volker Markl. 2015. Self-Tuning, GPU-Accelerated Kernel Density Models for Multidimensional Selectivity Estimation. In *Proc. of the 2015 ACM SIGMOD*. ACM, NY, USA, 1477–1492.
- [18] Yannis E. Ioannidis. 2003. The History of Histograms (abridged). In *VLDB*. Morgan Kaufmann, San Francisco, CA, USA, 19–30.
- [19] J. B. Kruskal. 1964. Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis. *Psychometrika* 29, 1 (1964), 1–27.
- [20] Joseph B Kruskal. 1964. Nonmetric multidimensional scaling: a numerical method. *Psychometrika* 29, 2 (1964), 115–129.
- [21] Hongrae Lee, Raymond T Ng, and Kyuseok Shim. 2011. Similarity join size estimation using locality sensitive hashing. *Proceedings of the VLDB Endowment* 4, 6 (2011), 338–349.
- [22] V. Levenshtein. 1966. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady* 10, 8 (1966), 707–710.
- [23] D. O. Lofsgaarden and C. P. Quesenberry. 1965. A Nonparametric Estimate of a Multivariate Density Function. *Ann. Math. Statist.* 36, 3 (06 1965), 1049–1051.
- [24] Yossi Matias, Jeffrey Scott Vitter, and Min Wang. 1998. Wavelet-based Histograms for Selectivity Estimation. *SIGMOD Rec.* 27, 2 (June 1998), 448–459.
- [25] S. Muthukrishnan, Viswanath Poosala, and Torsten Suel. 1999. On Rectangular Partitionings in Two Dimensions: Algorithms, Complexity, and Applications. In *ICDT*, Vol. 1540. Springer, London, UK, 236–256.
- [26] Emanuel Parzen. 1962. On Estimation of a Probability Density Function and Mode. *Ann. Math. Statist.* 33, 3 (09 1962), 1065–1076.
- [27] H. Sagan. 1994. *Space-filling curves*. Springer.
- [28] Hanan Samet. 2006. *Foundations of multidimensional and metric data structures*. Morgan Kaufmann.
- [29] Stefan Schmitt, Daniel Kuhn, and Gerhard Klebe. 2002. A new method to detect related function among proteins independent of sequence and fold homology. *Journal of molecular biology* 323, 2 (2002), 387–406.
- [30] David W Scott. 2015. *Multivariate density estimation: theory, practice, and visualization*. John Wiley & Sons.
- [31] George R. Terrell and David W. Scott. 1992. Variable Kernel Density Estimation. *The Annals of Statistics* 20, 3 (1992), 1236–1265.
- [32] Jeffrey Scott Vitter. 1985. Random Sampling with a Reservoir. *ACM Trans. Math. Softw.* 11, 1 (1985), 37–57.
- [33] Yan Zheng, Jeffrey Jests, Jeff M. Phillips, and Feifei Li. 2013. Quality and Efficiency for Kernel Density Estimates in Large Data. In *Proceedings of the 2013 ACM SIGMOD (SIGMOD '13)*. ACM, New York, NY, USA, 433–444.

GeoAlign: Interpolating Aggregates over Unaligned Partitions

Jie Song
University of Michigan
Ann Arbor, Michigan
jiesongk@umich.edu

Murali Mani
University of Michigan, Flint
Flint, Michigan
mmani@umflint.edu

Danai Koutra
University of Michigan
Ann Arbor, Michigan
dkoutra@umich.edu

H. V. Jagadish
University of Michigan
Ann Arbor, Michigan
jag@umich.edu

ABSTRACT

Answering crucial socioeconomic questions often requires combining and comparing data across two or more independently collected data sets. However, these data sets are often reported as aggregates over data collection units, such as geographical units, which may differ across data sets. Examples of geographical units include county, zip code, school district, etc., and as such, they can be *incongruent*. To be able to compare these data, it is necessary to realign the aggregates from the source units to a set of target spatially congruent geographical units. Existing intelligent areal interpolation/realignment methods, however, make strong assumptions about the spatial properties of the attribute of interest based on domain knowledge of its distribution. A more practical approach is to use available reference data sources to aid in this alignment. The selection of the references is vital to the quality of prediction.

In this paper, we devise GeoAlign, a novel multi-reference crosswalk algorithm that *estimates* aggregates in desired target units. GeoAlign is adaptive to new attributes with need for neither distribution-related domain knowledge of the attribute of interest nor knowledge of its spatial properties in Geographic Information System (GIS). We show that GeoAlign can easily be extended to perform aggregate realignment in multi-dimensional space for general use. Experiments on real, public government datasets show that GeoAlign achieves equal or better accuracy in root mean square error (RMSE) than the leading state-of-the-art approach without sacrificing scalability and robustness.

1 INTRODUCTION

Data are often found in silos, created independently. For example, administrative agencies and governments collect a great deal of data about their domain, most of which are then published in aggregate form. The primary purpose of the data collection is administrative, and the choice of data representation and structure is made by each agency for its own purpose. These data can be invaluable for understanding many social issues, particularly in conjunction with other data sources. However, most administrative agencies are not concerned with interoperability with other agencies, therefore standardization is unlikely. On the other hand, agencies value the privacy of individual citizens, and do not want any benefits from public data release to hurt their primary administrative mission. Therefore, in many cases, they

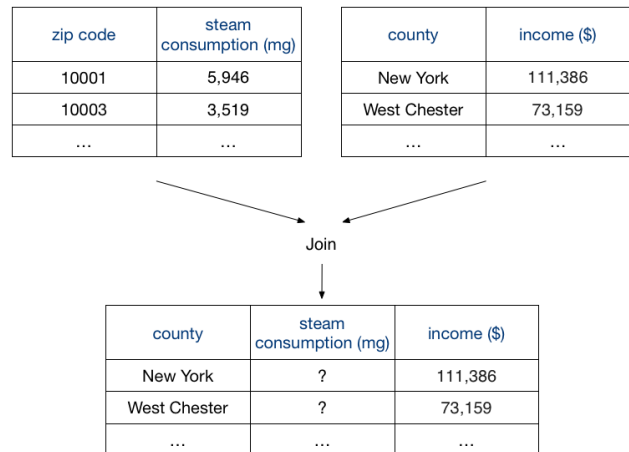


Figure 1: Join two tables for steam consumption (mg) and per capita income (\$) in New York State together by county.

will release data only in aggregate form. Similar reasoning applies in many other contexts as well. For example, Google Trends data is aggregated by geographical unit and time period, to avoid disclosing information about individual queries.

Data integration [25, 34] has been extensively studied, since there is often great benefit from joining multiple data sets. The bulk of the work on this topic addresses structural discrepancies, through schema mapping [2, 26, 42], and identification of individuals across data sets, through entity matching [21, 29]. One challenge not addressed in data integration is the case of data reported as aggregates over incompatible geographical/temporal units. This is a practical problem faced by government data center, NGOs, social scientists, and the general public when trying to related socioeconomic data to drive decision making processes, approximately 80% of which are related to a geographical location [14]. Even if the intention of joining such aggregated data based on their spatial or temporal properties seems to be the reasonable action of practice, these aggregates cannot easily be realigned accurately.

Motivating example. Let us consider two tables shown in Figure 1 – one table has the steam consumption amount aggregated by zip code and the other has the per capita income reported by county. A sociologist wants to study the correlation of energy consumption with income in order to plan for future energy supply arrangement. Valuable insight could be obtained by joining these two tables. However, this is not straightforward since the data are reported on incompatible aggregate units, since one zip code may

intersect several counties and one county may contain or overlap with multiple zip codes.

This challenge can be addressed by realigning one or both data sets to a common geographic type (target type) before performing the join. Let the intended target type be county, by which the per capita income is already reported. However, we only know the steam consumption amount by zip code, and have to estimate the number for each county. This estimate is obtained as a form of interpolation. Finding a good estimate of steam consumption per county is the challenge we need to address.

This problem of estimating aggregate values for geographic areas arises in many contexts, and has been extensively studied. *Areal Interpolation*, in Geographical Information Systems (GIS), is the process of aligning an attribute from one areal unit system (the source type of a set of polygons) to another spatially incongruent system (the target type of another set of polygons) [12, 22, 23, 31, 33]. It is more commonly known as *crosswalk*, or the *modifiable areal unit problem* in socioeconomic fields. If the attribute is uniformly distributed in space, then the interpolation can be performed in a straightforward way based on area. For example, if 70% of the area of a zip code lies in county A and 30% in county B, then we could estimate that 70% of the crimes reported in the zip code occurred in county A and the remaining 30% in B.

This uniform distribution assumption or homogeneity assumption rarely holds in practice. If we know something about the distribution, that can be taken into account in the interpolation. For example, if we know that more crimes occur in densely populated urban areas than in sparsely populated rural areas, we can take this into account. The mathematics can be tricky depending on exactly what we know about the distribution of the attribute of interest, so there has been a stream of research in the literature towards solving the problem based on different assumptions. We discuss this more in the related work.

In the data integration scenario, we often do not know much about an attribute of interest. Therefore, we may be unable to develop good rules for how it should be distributed. Even so, we can do better than make an unrealistic uniformity assumption, if we have access to additional data. In particular, if we can find a *reference* attribute, for which we know the detailed distribution, we can use it to perform a crosswalk from source units to target units of aggregation. For example, we may have detailed distribution available for population, with fine granularity aggregates giving us the population in every intersection of county and zip code. If we believe the crimes are distributed similarly to population (or at least more similarly to population than to area), then we can exploit our knowledge of population distribution to estimate the desired aggregates for number of crimes. In particular, consider a zip code with a population of 25,000 people. Suppose this zip code intersects two counties A and B, with the population in the intersections being 10,000 and 15,000 respectively. Suppose that we know there were 100 reported crimes in this zip code last year. We can estimate that 40 of these crimes occurred in county A and 60 occurred in county B, following the same ratio as the population. This approach makes no assumptions about the probability distribution of the reference attribute or the attribute of interest. It can work well if the attribute of interest is distributed similarly to the reference attribute. To the extent the distributions differ, the estimates will be off.

In this paper, our goal is to solve this data alignment problem through the use of more data. We often may have access to

more than one candidate reference attribute, each with its own distribution. We may not have domain knowledge enough to understand which reference is most similar to our variable of interest. Even if we found the best reference, its distribution may still not be close enough. Is there some way we can combine the information in the multiple reference attributes to do better? And at the same time, more adaptively predicts the estimates to new attributes of interest than using a single reference.

In this paper, we develop `GeoAlign`, a technique that does just this. The idea is to weight their relative contributions to the final estimate so that the most similar reference attributes have the greatest impact on the estimate.

The intellectual contributions of the paper are as follows:

- We define the general aggregate interpolation problem over unaligned partitions in one or more dimensions, which is an important problem in data integration (§2).
- We propose `GeoAlign`, an adaptive multi-reference crosswalk algorithm that solves the areal interpolation problem by realigning aggregates from source units to target units by learning distribution similarities between the attribute of interest and the reference attributes (§3). We show that `GeoAlign` can be used not just in two-dimensional maps but also for spaces with arbitrary numbers of dimensions.
- We evaluate the performance of `GeoAlign` against real data from `data.ny.gov` and Esri data in 2-dimensional space. These experiments show that `GeoAlign` outperforms the state-of-the-art single reference crosswalk approach in accuracy (§4). It is, at the same time, efficient, scalable and robust to noisy references even when limited references are available.

We then survey related work in areal interpolation (§5) before we conclude with future work (§6).

2 PROBLEM STATEMENT

In this section, we first introduce the terms we use throughout this paper before we formally define the aggregate interpolation problem in multi-dimensional space. We then illustrate, with examples, the aggregate interpolation problem in 2-D and in other dimensions.

2.1 Preliminaries

In Geometry, an n -dimensional universe $\Omega \subset \mathbb{R}^n$ can be partitioned into some *unit system* γ^y composed of a set of *units* $U^y = \{u_1^y, u_2^y, \dots\}$, where $\forall u_i^y \in U^y, u_i^y \subset \mathbb{R}^n$. Units in U^y satisfy

$$\forall u_i^y, u_j^y \in U^y, i \neq j, u_i^y \cap u_j^y = \emptyset, \quad (1)$$

that is any pair of units in U^y is disjoint with each other since they have no spatial overlap in n dimensions. Suppose that an attribute of interest α_x exists, then we denote its *aggregate vector* as $a_x^y = [a_x^y[1], a_x^y[2], \dots, a_x^y[|U^y|]]$ such that $a_x^y[i]$ is the aggregate of α_x in the i th unit of U^y .

As an example in 2-D space, in the universe of New York State Ω , county partitions compose a unit system γ^y . They share no areal intersection such that they are spatially incongruent with each other. Steam consumption, which is the attribute of interest α_x , has its data in Figure 1 collected from such a set of county units U^y . Another possible unit system is zip code partitions. We can view the steam consumption column in the table as its aggregate vector a_x^y for the county unit system. Each entry of

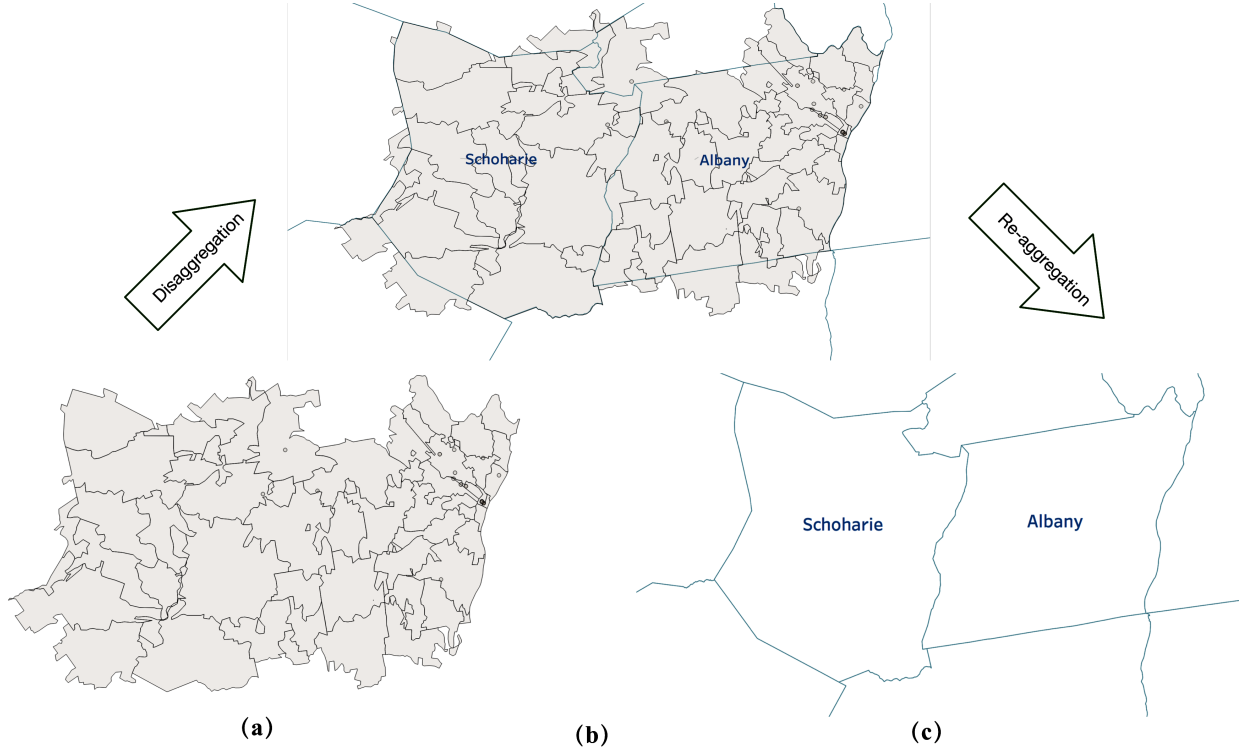


Figure 2: Examples of units in the partial map of New York State for aggregate interpolation: (a) zip code units (source units), (b) zip code and county intersection units and (c) county units (target units).

the vector represents the amount of steam consumption in some county.

2.2 The Aggregate Interpolation Problem

We define the following terms for the aggregate interpolation problem in \mathbb{R}^n :

- $U^s = \{u_1^s, u_2^s, \dots\}$, source units of the source unit system γ^s in the universe Ω .
- $U^t = \{u_1^t, u_2^t, \dots\}$, target units of the target unit system γ^t in the same universe.
- $a_o^s = [a_o^s[1], a_o^s[2], \dots, a_o^s[|U^s|]]$, aggregate vector of the objective attribute α_o in source units. $a_o^s[i]$, the i^{th} aggregate of a_o^s , is collected from source unit u_i^s .
- $a_o^t = [a_o^t[1], a_o^t[1], \dots, a_o^t[|U^t|]]$, aggregate vector of the objective attribute α_o in target units. $a_o^t[j]$, the j^{th} aggregate of a_o^t , is collected from target unit u_j^t .

Given U^s, U^t and a_o^s , aggregate interpolation approximates a_o^t as $\hat{a}_o^t = [\hat{a}_o^t[1], \hat{a}_o^t[2], \dots, \hat{a}_o^t[|U^t|]]$.

Aggregate Interpolation Problem in 2-D When it comes to a 2-dimensional space \mathbb{R}^2 , units are *simple polygons* consisting of straight, non-intersecting edges forming a closed path by pairwise join. A unit in 2-dimensional space can be denoted by

$$u_i = (V_{u_i}, E_{u_i}) \text{ where } |V_{u_i}| = |E_{u_i}| = n_i, \quad (2)$$

where V_{u_i} is a set of vertices in \mathbb{R}^2 and E_{u_i} is a set of edges connecting the vertices in V_{u_i} such that every vertex is shared by exactly two edges. Then, u_i is the closed area formed by connecting n_i vertices in V_{u_i} by n_i edges in E_{u_i} .

This problem is referred to as the *areal interpolation* problem in the GIS community. The 2-dimensional space is the map; and the

unit system, also recognized as feature layer in GIS, is composed of partitions delimited by boundaries of some geographic type. Some of the most widely used geographic types in demographic data are county, zip code, and more. For instance, as shown in Figure 2, U^s is the feature layer for zip code in (a); U^t is the other feature layer for counties in (c). Given the aggregates of steam consumption in zip codes a_o^s shown in Figure 1 from the motivating example, the aggregate interpolation problem in 2-D approximates the steam consumption in counties, \hat{a}_o^t .

Aggregate Interpolation Problem in other dimensions In the 1-dimension setting of the problem, units are *intervals* or *line segments* between two points such that

$$u_i = [u_{i_1}, u_{i_2}], \quad (3)$$

where u_{i_1} and u_{i_2} are two points on the real line \mathbb{R} . We may illustrate the problem as interpolation of population histogram aggregates for two sets of age intervals as depicted in Figure 3. In this case, we can treat the set of narrow bins of age in (a) as U^s , the set of wide bins of age in (b) as U^t , for the same range of age as the universe of interest Ω . Given the population histogram for narrow age bins, a_o^s , the aggregate interpolation problem in 1-D predicts the population histogram for wide age bins \hat{a}_o^t .

Unit system overlapping also exist in 3-D or higher dimensions. One example is 3-D GIS data, such as the distribution of disease, evaluated for cubic units of different size scales. Another example is the data collected for 4-D space (3D) and time systems, such as environmental exposures, crosswalked to another system incongruent in both space and time units. For both cases, areal interpolation is the bridge to map the data across unit systems to enable side-by-side comparison with data from incompatible units.

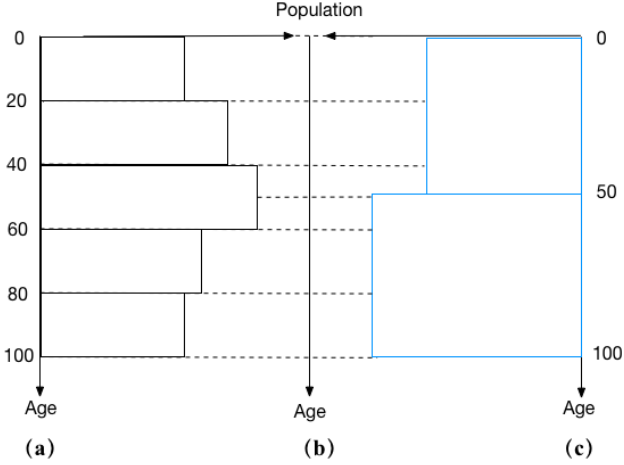


Figure 3: Realign population histogram in two sets of age intervals by transforming aggregates from (a) narrow bins to (c) wide bins. The dotted lines separate the age range into a set of tentative intersection units as in (b).

3 AGGREGATE INTERPOLATION BY GEOALIGN

In this section, we first introduce some additional definitions and notations used throughout the rest of the paper and a general two-step solution solving the aggregate interpolation algorithm. We then lay the groundwork for the assumptions made by GeoAlign before exploring the details of the algorithm.

3.1 GeoAlign preliminaries

Before introducing the general steps to solve the aggregate interpolation problem, we further define the set of *intersection units* for the intersection unit system γ^{st} as $U^{st} = \{u_1^{st}, u_2^{st}, \dots\}$, where $\forall u_k^{st} \in U^{st}, u_k^{st} \subset \mathbb{R}^n$. Each intersection unit is a subregion within some source unit and some target unit, that is

$$\forall u_k^{st} \in U^{st}, \exists u_i^s \in U^s \wedge u_j^t \in U^t, u_k^{st} \subseteq u_i^s \text{ and } u_k^{st} \subseteq u_j^t. \quad (4)$$

It can be thus deduced that $|U^{st}| \geq \max(|U^s|, |U^t|)$.

The aggregate vector of the intersection units for some attribute α_x is denoted as $a_x^{st} = [a_x^{st}[1], a_x^{st}[2], \dots, a_x^{st}[|U^{st}|]]$.

In the simplest case, the intersection units are the n -dimensional spatial intersections of source and target units. For instance, for the areal interpolation problem in Figure 2, U^{st} is the set of intersection areas between zip codes and counties in (b); and for the histogram realignment problem in Figure 3, U^{st} is the set of age intersection intervals between source and target bins. More fine-grained partitions of intersection units may be introduced if necessary when disparate spatial properties of the attribute in these partitions are introduced by auxiliary data.

Assume that the probability density function of attribute α_x for γ^{st} is a piecewise function, denoted as

$$f_x^{st}(z) = \begin{cases} f_x^{st}[1](z) & , z \subset u_1^{st} \\ f_x^{st}[2](z) & , z \subset u_2^{st} \\ \dots & \\ f_x^{st}[|U^{st}|](z) & , z \subset u_{|U^{st}|}^{st} \end{cases} \quad (5)$$

is known, then its aggregate in the source units and target units follows:

$$\begin{aligned} a_i^s &= \sum_{\forall u_k^{st} \in U^{st}, u_k^{st} \subseteq u_i^s} a_k^{st} \\ &= \sum_{\forall u_k^{st} \in U^{st}, u_k^{st} \subseteq u_i^s} \int_{z \subset u_k^{st}} f_x^{st}[k](z) dz, \end{aligned} \quad (6)$$

and similarly

$$\begin{aligned} a_j^t &= \sum_{\forall u_k^{st} \in U^{st}, u_k^{st} \subseteq u_j^t} a_k^{st} \\ &= \sum_{\forall u_k^{st} \in U^{st}, u_k^{st} \subseteq u_j^t} \int_{z \subset u_k^{st}} f_x^{st}[k](z) dz. \end{aligned} \quad (7)$$

Alternatively speaking, the aggregate in each source/target unit is equivalent to the sum of aggregates of all intersection units within it.

Two-step Approximation. We use a two-step solution to solve the aggregate interpolation problem for objective attribute α_o . In our solution, we first compute the approximate a_o^{st} (a_o^{st} is the aggregate vector for the intersection units). We then aggregate these approximate intersection unit aggregates to determine the approximate target unit aggregates. The two steps in our solution are described below:

- (1) **Disaggregation:** Split the aggregates in each source unit to its intersection units. Mathematically speaking,

$$\hat{a}_o^{st}[k] = \mathcal{B}(a_o^s[i], \dots), \text{ s.t. } u_k^{st} \subseteq u_i^s, \quad (8)$$

where the disaggregation function $\mathcal{B}(a_o^s[i], \dots)$ computes the approximated $\hat{a}_o^{st}[k]$ of $a_o^{st}[k]$. Note that \dots denotes the ancillary data that contribute to the approximation. Some of the most commonly used ancillary data are shape files of u_i^s and u_k^{st} , etc. More advanced approximation function may use external ancillary data. For instance, the distribution of a reference attribute that is positively related to the distribution of α_o .

- (2) **Re-aggregation:** Aggregate the approximated intersection unit aggregates for the target unit they reside in, or equivalently

$$\hat{a}_o^t[j] = \sum_{\forall u_k^{st} \in U^{st}, u_k^{st} \subseteq u_j^t} \hat{a}_o^{st}[k]. \quad (9)$$

General Solution Properties. Regardless of the types of ancillary data available, some constraints are widely adopted in the existing two-step approximation solutions. We name two of them here.

One of these constraints is the *volume preserving* property [31, 46]. This property ensures that every source aggregate is preserved by the total of approximated aggregates in its intersection units, or

$$a_o^s[i] = \sum_{\forall u_k^{st} \in U^{st}, u_k^{st} \subseteq u_i^s} \hat{a}_o^{st}[k]. \quad (10)$$

The property is improving the estimation in that greater fidelity is given to the approximation in the intersection units, which propagates to a more accurate estimation in target units. It has been shown experimentally that methods following the volume preserving property make comparatively better predictions [31].

Homogeneity is also often used to compensate for the absence of information. Mathematically, for some attribute α_x , its probability density function in a given unit is constant. In other words,

its aggregate on any sub-unit of the given unit is proportional to the area of the sub-unit. However, the assumption of homogeneity is rarely met in the real world [49].

3.2 GeoAlign Assumptions

We often have access to multiple reference attributes, no one of which perfectly matches the objective attribute we wish to estimate. It would appear advantageous for us to use all of them instead of using a single reference attribute as the current extensive approaches described above. To this end, we propose GeoAlign, an aggregate interpolation algorithm that realigns aggregated data by learning from a combination of reference attributes to best predict the actual aggregates of the objective attribute in target units. GeoAlign leverages the advantages of extensive approaches and is, at the same time, robust to various objective attributes.

An intuitive idea could be to model the objective attribute aggregates as a function of multiple reference attributes aggregates in source units, evaluate coefficients with estimation methods and substitute reference attributes in target units for prediction. However, this is not applicable for the aggregate interpolation algorithm since training samples (objective attribute aggregates in source units) and test samples (objective attribute aggregates in target units) are not randomly drawn from the same population and the test samples are constrained by the training samples.

To address the linkage between two sets of samples and to account for the scale variations of reference attributes, in GeoAlign, the realignment of the objective attribute is related to that of the reference attributes through a statistical model for re-aggregation. In order to make the problem tractable, we assume that different attributes are independent across source units, and that every attribute is correlated in its distribution between source and target units. We will loose the independence assumption of references later as shown in experiments in §4.4.2.

3.3 Disaggregation Matrix

Since we study the partition of aggregates in intersection units, in the disaggregation step, $\mathcal{B}(a_o^s[i], \dots)$ can be reformulated as

$$\hat{a}_o^{st}[k] = \frac{\omega_o^{st}[k]}{\omega_o^s[i]} a_o^s[i]$$

subject to $\sum_{\forall u_k^{st} \in U^{st}, u_k^{st} \subseteq u_i^s} \omega_o^{st}[k] = \omega_o^s[i],$ (11)

where $\frac{\omega_o^{st}[k]}{\omega_o^s[i]}$ is the share of aggregate in the k -th intersection unit ($\omega_o^{st}[k]$) over that in the i -th source unit ($\omega_o^s[i]$) it resides in. Intuitively, the re-aggregation step sums up the weighted share of all intersection units in all source units that overlap with the target unit. Alternatively speaking,

$$\hat{a}_o^t[j] = \sum_{\forall u_i^s, u_i^s \cap u_j^t \neq \emptyset} \frac{\sum_{\forall u_k^{st} \subseteq u_i^s \cap u_j^t} \omega_o^{st}[k]}{\omega_o^s[i]} a_o^s[i].$$
 (12)

Rather than approximating a_o^{st} in the disaggregation step, we can instead infer $\frac{\omega_o^{st}[k]}{\omega_o^s[i]}$, $\omega_o^{st}[k]$ or $\sum_{\forall u_k^{st} \subseteq u_i^s \cap u_j^t} \omega_o^{st}[k]$. This choice often depends on the type of ancillary data available. The most widely used ancillary data is the true disaggregation of a reference attribute between source and target units. For instance, for the population reference mentioned in the introduction, the population aggregates in intersection units of counties and zip codes. We denote the *disaggregation matrix* of some attribute

Table 1: Notations in §2 and 3

Notation	Description
Ω	an n-dimensional universe of interest
γ^y	a unit system in Ω , for example γ^s at source level
$U^y = \{u_1^y, u_2^y, \dots\}$	the set of units in γ^y
α_o	the objective attribute
$A_r = \{\alpha_{r_1}, \alpha_{r_2}, \dots\}$	the set of reference attributes
$\alpha_x \in \alpha_o \cup A_r$	an attribute of interest
$a_x^y = [a_x^y[1], a_x^y[2], \dots, a_x^y[U^y]]$	the aggregate vector of α_x in units of U^y
f_x^y	the probability density function of α_x for γ^y
$\mathcal{B}(a_o^s[i], \dots)$	the disaggregation function
ω_x^y	the weighted share vector of α_x for γ^y
α_x^y	the normalized α_x^y
$DM_x^{y_1, y_2}$	the dimension matrix of α_x , where $DM_x^{y_1, y_2}[i, j]$ is the aggregate of α_x in the intersection of $u_i^{y_1}$ and $u_j^{y_2}$
$\beta = [\beta_1, \beta_2, \dots, \beta_{ A_r }]$	weights computed from Equation (15)

α_x between two unit systems γ^{y_1} and γ^{y_2} as $DM_x^{y_1, y_2}$, where $DM_x^{y_1, y_2}[i, j]$ is its aggregate in the intersection area of $u_i^{y_1}$ and $u_j^{y_2}$. For γ^s and γ^t ,

$$DM_x^{s, t}[i, j] = \sum_{\forall u_k^{st} \subseteq u_i^s \cap u_j^t} a_x^{st}[k] \quad (13)$$

The disaggregation matrix of the reference attribute between source and target units is often wrapped up in a crosswalk relationship file. When the disaggregation matrix of only one reference attribute α_r is available, we can substitute $\sum_{\forall u_k^{st} \subseteq u_i^s \cap u_j^t} \omega_o^{st}[k]$ for $DM_r^{s, t}$ to complete the approximation of the objective attribute in target units. This type of method is named as the *dasymetric method* [32, 33, 48]. A special case of it is the areal weighting method [30], using the disaggregation matrix of area as the reference. Dasymetric methods are widely employed in socioeconomic data realignment by general users [10].

Since we only consider the disaggregation matrix between source and target units, from now on, we use DM_x for $DM_x^{s, t}$.

3.4 GeoAlign Algorithm

In the real world, the disaggregation matrix of more than one references attributes is often available. GeoAlign is a volume-preserving method that leverages the distribution similarity of the objective attribute with reference attributes at the source level and predicts the dimension matrix of the objective as a weighted combination of the dimension matrices of the references. We will first extend some of the notations in Section 2, and then describe our proposed algorithm in detail.

Notation. Let $A_r = \{\alpha_{r_1}, \alpha_{r_2}, \dots\}$ be the set of reference attributes available. The aggregate vectors of these reference attributes in source units are represented as $a_{r_1}^s, a_{r_2}^s, \dots, a_{r_{|A_r|}}^s$, where $a_{r_k}^s = [a_{r_k}^s[1], a_{r_k}^s[2], \dots, a_{r_k}^s[|U^s|]]$ for the k th reference attribute. Similarly, the aggregate vectors of these reference attributes in target units are represented as $a_{r_1}^t, a_{r_2}^t, \dots, a_{r_{|A_r|}}^t$, where $a_{r_k}^t = [a_{r_k}^t[1], a_{r_k}^t[2], \dots, a_{r_k}^t[|U^t|]]$.

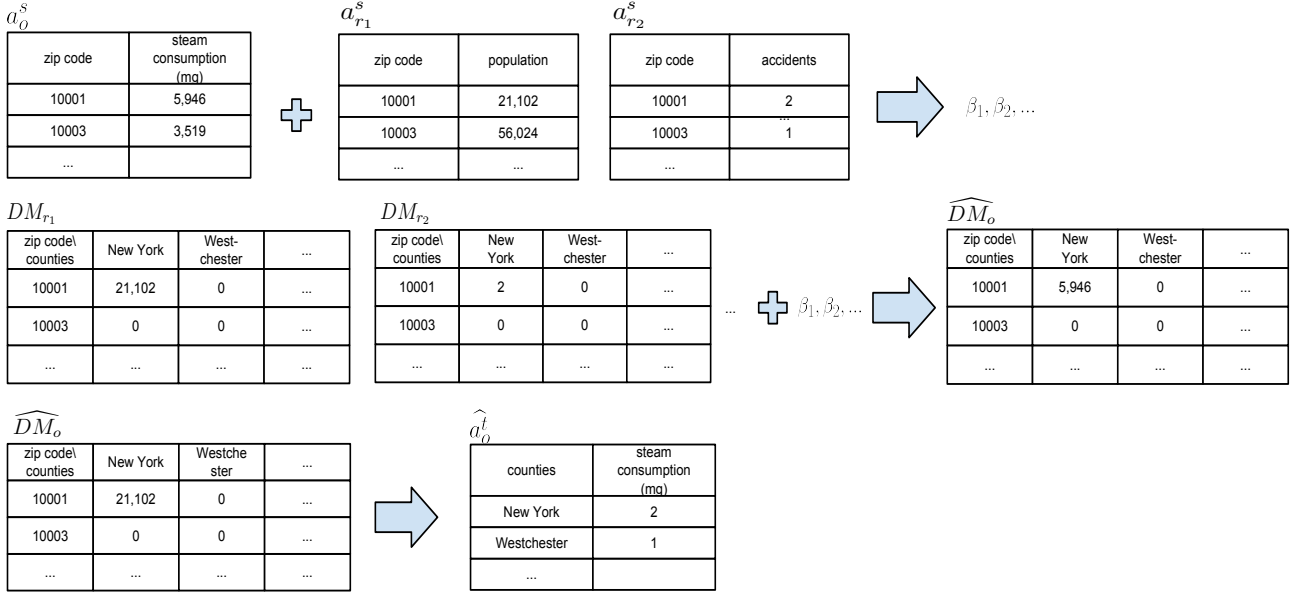


Figure 4: GeoAlign interpolation for the objective steam consumption data in Figure 1 from zip codes to counties using two reference attributes: population and accidents, in three steps: weight learning, disaggregation and re-aggregation.

We assume that the ancillary data available is the disaggregation matrix of all the reference attributes. We denote the disaggregation matrix of the k th reference attribute as DM_{r_k} .

To avoid variation in scale, we normalize the objective attribute and the references at the source level, adjusting their values measured on different scales to a notionally common scale. This is reasonable in two ways. First, GeoAlign is dependent on the distribution similarity between the objective attribute and the references across source units rather than their actual value similarity. Second, GeoAlign jointly considers the similarity of the objective with multiple references. The magnitude of the references should not be a contributing factor.

The normalized $a_{r_k}^s$ is denoted by $a_{r_k}^{s'}$ for $k = 1, 2, \dots, |A_r|$, and is computed as $a_{r_k}^{s'} = a_{r_k}^s / \max_{i, i \in |U^s|} a_{r_k}^s[i]$, $a_{r_k}^{s'}[i] \geq 0$.

The aggregate vector of the objective attribute in source units a_o^s is also normalized similarly, and is denoted as $a_o^{s'}$.

GeoAlign Steps In the disaggregation step, GeoAlign computes \widehat{DM}_o , which is the estimated weighted dimension matrix of the objective attribute. Our intention is to best predict \widehat{DM}_o , and at the same time, preserve its volume preserving property. We propose

$$\widehat{DM}_o[i, j] = \begin{cases} \frac{\sum_{k=1}^{|A_r|} \beta_k \times DM_{r_k}[i, j]}{\sum_{k=1}^{|A_r|} \beta_k \times a_{r_k}^s[i]} \cdot a_o^s[i], & \sum_{k=1}^{|A_r|} a_{r_k}^s[i] \neq 0 \\ 0, & \text{otherwise} \end{cases} \quad (14)$$

where $\beta = [\beta_1, \beta_2, \dots, \beta_{|A_r|}]$ is the learned weight vector and $\sum_{i=1}^{|A_r|} \beta_i = 1$.

Our preliminary experiments lay the ground work of our assumption such that the higher the similarity between two attributes at the source level, the more likely their distribution in the intersection level are similar. We can thus express the objective attribute as linearly associated with the reference attributes for both aggregate vector in source units and disaggregation matrix. The weights are obtained by solving a constrained linear

least squares programming problem with objective function:

$$\begin{aligned} & \min_{\beta} \frac{1}{2} \|A\beta - \mathbf{b}\|^2 \\ & \text{subject to } \sum_{k=1}^{|A_r|} \beta_k = 1 \\ & \text{where } \beta_k \geq 0, \text{ for } k = 1, 2, \dots, |A_r| \end{aligned} \quad (15)$$

where A is the column-wise concatenation of $a_{r_k}^{s'}$ for $k = 1, 2, \dots, |A_r|$ and \mathbf{b} is $a_o^{s'}$. Instead of computing \widehat{DM}_o by directly applying the weights to DM_{r_k} s, we adapt it to the scale of reference attributes and insert back the weights to Eq. (14) to get an adjusted \widehat{DM}_o .

The approximated disaggregation matrix of the objective attribute satisfies the volume preserving property such that

$$\widehat{DM}_o[i, j] \geq 0 \quad \text{and} \quad \sum_{j=1}^{|U^t|} \widehat{DM}_o[i, j] \approx a_o^s[i]. \quad (16)$$

The estimated aggregates of the objective attribute in target units are computed in the reaggregation step as

$$\widehat{a}_o^t[j] = \sum_{i=1}^{|U^s|} \widehat{DM}_o[i, j] \quad (17)$$

Following the pseudocode in Algorithm 1, we further illustrate the algorithm by the motivating example in Figure 1, with the steps depicted in Figure 4. Assume that GeoAlign is crosswalking the steam consumption objective from zip codes to counties. Moreover, assume that the aggregate vectors, $a_{r_1}^s$ and $a_{r_2}^s$, and the disaggregation matrices, DM_{r_1} and DM_{r_2} , for two reference attributes, population and accidents, are readily available. Maximizing the distribution similarity across units between the normalized objective, $a_o^{s'}$, and the normalized references, $a_{r_1}^{s'}$ and $a_{r_2}^{s'}$, the objective attribute is first optimized as a weighted combination of the references at the source level (zip code level). The weights, β_1 and β_2 , are then reassigned to the disaggregation matrices of the references DM_{r_1} and DM_{r_2} , and adjusted to predict an approximated disaggregation of the objective \widehat{DM}_o . The approximated disaggregation matrix is eventually re-aggregated

Algorithm 1: GeoAlign

Input: aggregate vectors of reference attributes in source units $a_{r_1}^s, a_{r_2}^s, \dots, a_{r_{|A_r|}}^s$; corresponding disaggregation matrices $DM_{r_1}, DM_{r_2}, \dots, DM_{r_{|A_r|}}$; and the aggregate vector of the objective attribute in source units a_o^s .

Output: estimated aggregates of the objective attribute in target units \hat{a}_o^t

- 1 **Step 1. Weight Learning:** Compute weights, β , by solving the least squares problem in Equation (15)
 - 2 **Step 2. Disaggregation:** Compute the estimated weighted disaggregation matrix of the objective attribute, \widehat{DM}_o , using Equation (14)
 - 3 **Step 3. Re-aggregation:** Re-aggregate to estimate the aggregates of the objective attribute in target units, \hat{a}_o^t , using Equation (17)
-

to derive an approximate of the objective at the target county level (\hat{a}_o^t).

It can be easily shown that GeoAlign is applicable to any dimension since the algorithm involves no dimension dependent information or computation. Rather, the only information needed is the true partition of reference attributes in source and target intersection units regardless of dimension or dimension-related information, such as spatial correlation for geospatial data. Alternatively, if true partition of references in finer granularity is available, the data can be aggregated to the level of source and target intersection as a reference attribute.

4 EXPERIMENTAL EVALUATION

We evaluated the feasibility of the GeoAlign algorithm from two crucial aspects: whether the algorithm can correctly complete the realignment task (effectiveness), and whether the runtime of the algorithm is fast enough (efficiency). Additionally, we consider runtime scalability when larger datasets are involved and the robustness of the algorithm when low quality or limited reference attributes present.

We compare the performance of GeoAlign with that of areal weighting method [31] and dasymetric method [32, 33, 48] that utilizes three reference attributes separately.

4.1 Experimental Setup

We developed the GeoAlign algorithm in Python. All experiments were performed on a 2.3 GHz Intel Core i7 with 8 GB memory and a 7200 rpm SATA disk.

We evaluated GeoAlign for 2-D areal interpolation. We used county and zip code as the two geographic types of interest, and focused on data from two different universes, New York State and the United States. Most of the New York State data are collected from `data.ny.gov`, populated in tabular form. Three population level demographic datasets have been used as reference data for the single crosswalk algorithm, namely the population data from United States Census Bureau [4], the aggregated USPS residential address data and the aggregated USPS business address data [41]. In addition, we also selected five large individual level datasets (The New York State Restaurants dataset is generated by selecting unique restaurants in the Food Service Inspections dataset) with geographic information and aggregated their number of records for the intersection area of the two geographic types to form

their disaggregation matrices [5–8]. Thus we obtained a total of eight reference datasets with accurate distributions by zip code and by county, and their disaggregation matrices from zip codes to counties.

Besides the three population level Census data, which cover the entire nation including New York State, other data for the United States were collected from Esri, where the Maps and Data group provides publicly available geocoded GIS data. Six individual level GIS data [15–20] were aggregated based on their geospatial information for zip code and county levels and their intersections using ArcGIS Pro [27]. We also computed the area of units at these three levels, which is later used as the reference attribute by the areal weighting method, yielding 10 datasets in total for the universe of the United States.

There are more datasets with attributes for which the aggregate vectors are available for both zip code and county for New York State or for the United States. However, we did not use them as reference attributes due to two reasons. First, it was not clear whether these aggregates are accurate or approximate. In §4.4.1, we further discuss the impact of the reference approximates on the prediction. The other reason is that several attributes do not have their disaggregation matrices publicly accessible and such attributes cannot be used as reference attributes. In case of limited reference attributes, we show in §4.4.2 that GeoAlign makes reasonable predictions even when the references are poorly selected.

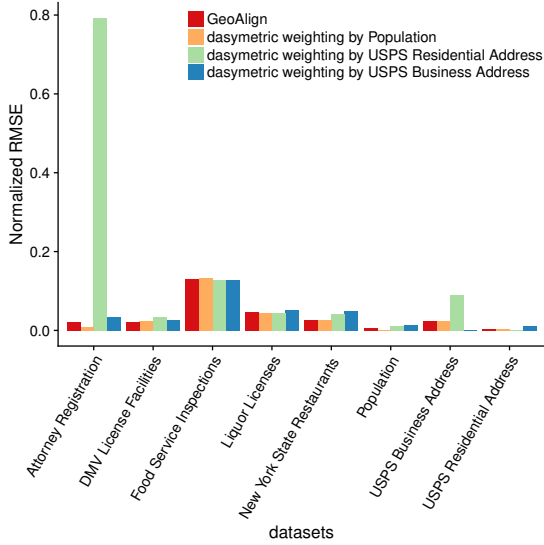
Since the number of datasets with accurate disaggregation matrix is limited, we adopted the cross-validation evaluation method that deals with the problem well. We conducted two series of experiments, one for each universe. More specifically, for each universe, we picked one of the datasets as the test dataset, in turn, and used the remaining datasets to develop crosswalks in GeoAlign whose combined weighted performance is then evaluated for the test dataset. The performance of GeoAlign is compared with the base-line single reference crosswalk method that redistributes by a disaggregation matrix of some known attribute. More specifically, GeoAlign is compared with the areal weighting method and the dasymetric algorithm referencing the three population level datasets. Note that when one of the population reference datasets or the area dataset is used as the test dataset, the performance of both methods referencing this dataset is not evaluated.

4.2 GeoAlign Effectiveness

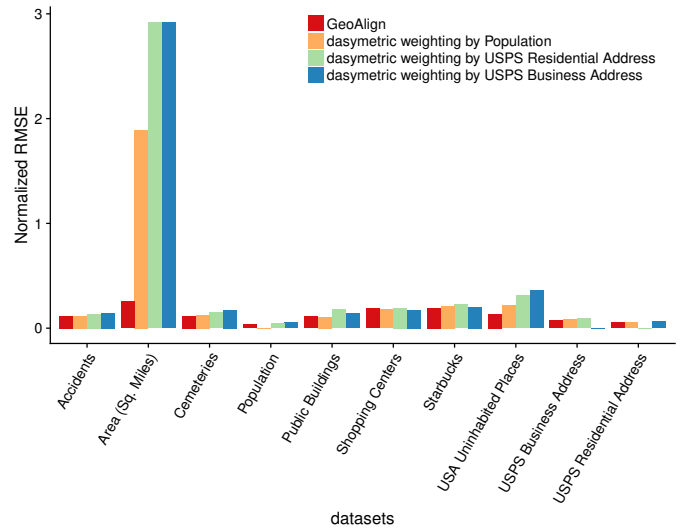
To evaluate the effectiveness of GeoAlign, we adopted root mean square error (RMSE) as the evaluation criterion that computes the deviation of estimated aggregates from true aggregates of the attribute in counties. To ease the comparison across datasets of heterogeneous scales, in Figure 5, we show the RMSE normalized by the mean of the measured data (NRMSE).

The NRMSE of GeoAlign is compared with that of the dasymetric method using three population level datasets and the areal weighting methods for both New York States (Figure 5a) and the United States (Figure 5b), using eight and ten datasets respectively. The performance of areal weighting method is not shown in the figure since it makes poor predictions for all test datasets: over 15 times of the NRMSE of GeoAlign for New York State experiments and over 50 times of the NRMSE of GeoAlign for the United States experiments.

The NRMSE of GeoAlign is less than 0.13 for New York State experiments and less than 0.26 for the United States experiments.



(a) New York State



(b) the United States

Figure 5: GeoAlign prediction performance (NRMSE) compared with dasymetric methods. Since a better prediction yields a lower NRMSE, GeoAlign is making comparable or better predictions than the dasymetric methods for tests in New York State and the United States.

Though three dasymetric methods have comparable error on most datasets, for these datasets, GeoAlign is making equal or better predictions. It should also be noted that no one of these three methods is predicting uniformly well for all datasets as GeoAlign does, in whichever universe. For instance, the dasymetric method referencing the population data presents much higher error than the other methods when predicting for attorney registration and USPS Business Address counts for counties in New York State; all three dasymetric methods fail in accuracy for both area and USA uninhabited places datasets in the United States.

Except the USPS business address dataset, the rest three are individual level datasets with limited number of observational units that are sparsely distributed in the universe. Also, they do not align well with demographic attributes as those in the areal weighting and dasymetric methods. We observe that GeoAlign accounts for sparsity and heterogeneous distributions with flexibility.

4.3 GeoAlign Efficiency and Scalability

We evaluated the efficiency of GeoAlign in terms of algorithm runtime. Apart from the horizontal efficiency comparison across cross-validated tests for a given universe, we also considered the scalability of GeoAlign runtime. This is realized by comparing GeoAlign efficiency vertically across the universes of different scales.

In addition to New York State and the United States, new universes were selected as a set of states whose boundaries are congruent with any other state in the universe. The selection is a greedy process that ensures the states in a universe are tightly connected from a geospatial perspective. These four new universes include Mid-Atlantic division and Northeast region defined by Census Bureau, states contained entirely in the Eastern Time Zone and all states excluding the ones in the Census West Region (non-West). They form a spatial coverage hierarchy preventing the inter-state influence of randomly selected universes.

Moreover, for factor control purpose, instead of collecting more datasets for new universes, for each universe, we subset the ten datasets covering the United States, keeping the entries collected from units within the universe as inputs.

To avoid random error, we averaged the runtime across ten trials for the cross-validated experiments in each universe.

Experimental results show that GeoAlign runtime is stable across experiments for the same universe. This is consistent with our claim that the complexity of GeoAlign is not related to the magnitude of the count data. The majority of the runtime, over 90%, is spent on computing the disaggregation matrix after the weights are estimated. Note that the aggregate vectors of the objective attribute in source geographic units has the same size for all the different datasets (the size is $|U^s|$). Similarly the aggregate vectors of the reference attributes in source geographic units are all of the same size (all of size $|U^s|$), the aggregate vectors of the reference attributes in target geographic units are all of the same size (all of size $|U^t|$). Further, all the disaggregation matrices are all of the same size as well. The reason for the minor difference in GeoAlign runtime for different datasets is because of the difference in the number of non-zero entries in the disaggregation matrix, which is stored as sparse matrix, of reference attributes. For the disaggregation matrix, sparse datasets, such as cemeteries, have less non-zero entries, while dense datasets, such as population, have more non-zero entries. Matrix operations involving sparse matrices are influenced by this factor in SciPy package.

As for cross-universe comparison, we plotted GeoAlign runtime versus the number of zip codes (source units) and the number of counties (target units) in Figure 6. These two plots show that GeoAlign is fast: it runs for less than 0.15 second even for cross-walk between 30238 zip codes and 3142 counties in the United States universe. They also prove the linear relationship between GeoAlign runtime with the number of units in source and target levels since the dominating disaggregation matrix construction operation is linearly related to these two factors.

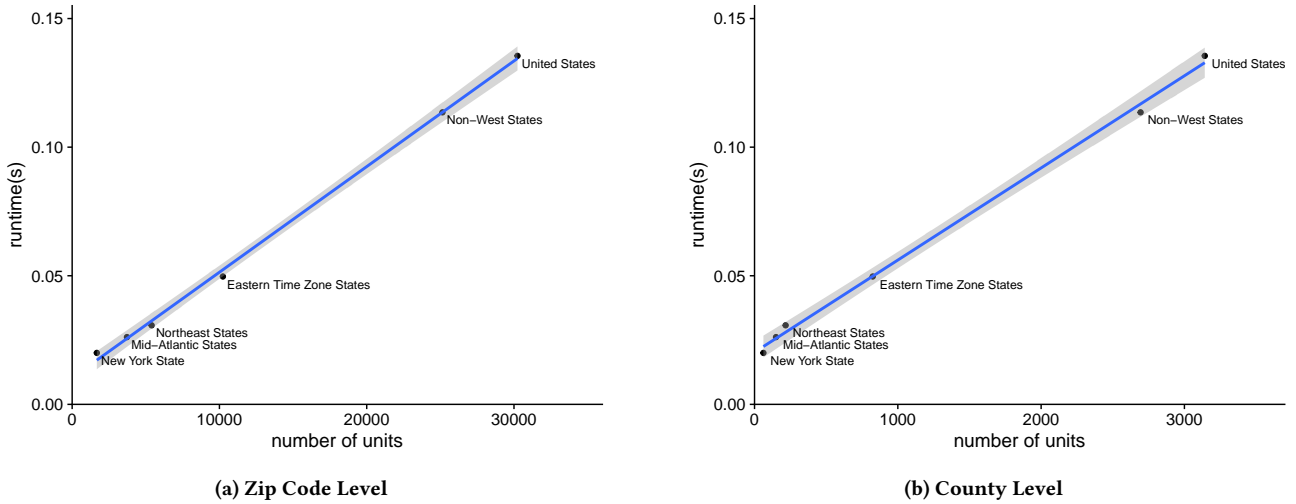


Figure 6: GeoAlign runtime scales linearly with respect to the number of units in source level and target level

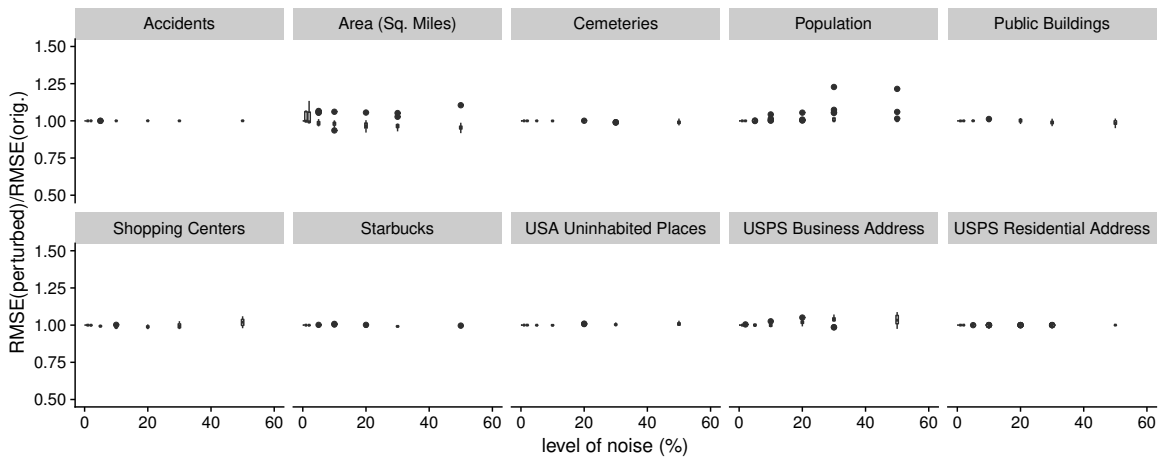


Figure 7: When noises are introduced in references, the prediction deviation is evaluated as the ratio of the RMSE using the perturbed references to the RMSE using the original references. The closer the ratio is to 1, the more invariant GeoAlign is to reference noises. For up to 50% level of noise, most experiments have the prediction deviation around 1 indicating the robustness of GeoAlign to noisy references.

4.4 GeoAlign Robustness

As mentioned earlier in §4.1, during the reference attribute collection process, we encountered two difficulties: the undetermined accuracy of reference attributes at the source level, and the limited availability of datasets with disaggregation matrix. We conducted two series of experiments evaluating the robustness of GeoAlign with respect to these two problems respectively.

4.4.1 Inaccurate Reference Attributes. Public aggregated data can be derived in multiple ways. They can be aggregates of individual level data, approximates derived from some crosswalk algorithm, etc. Without the raw data and the transformation information available, the accuracy of these aggregates are unknown. It is thus hard to determine whether the data can be used as references.

To quantitatively evaluate the influence of the accuracy of reference attributes on GeoAlign, we artificially introduced "noise" to the reference attributes. We define *noise* as the deviation from the actual value. Noise is measured by "levels" such that a $x\%$ level of noise for y is $\pm x * y/100$. The noise-polluted y is thus

$(1 + x/100) * y$. For each of the ten cross-validated experiments in United States, we synthetically generated noisy reference attributes at the source level with 1%, 2%, 5%, 10%, 20%, 30% and 50% degrees of noises for all references. Each experiment is replicated 20 times to account for random error due to randomness of positive or negative noises. We quantify the prediction deviation as the ratio of the RMSE using perturbed noisy reference attributes to the RMSE using the original reference attributes. The closer the prediction deviation is to 1, the smaller the impact of the noises is. GeoAlign is making better prediction with the perturbed reference attributes if the ratio is higher than 1; whereas a less than 1 ratio indicates worse prediction with the perturbed reference attributes.

In Figure 7, we show the box plot of the prediction deviation with respect to different levels of noise. The prediction performance of GeoAlign is stable across experiments. For each experiment, GeoAlign is making robust predictions for all levels of noise. Though for the area and population datasets, higher levels

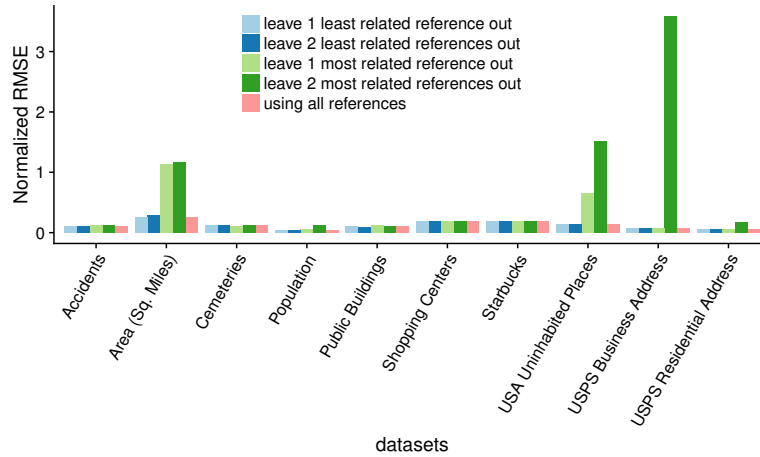


Figure 8: GeoAlign is robust to the choice of reference attributes. Though extra reference attributes do not create any loss, reference attributes with higher correlation with the objective are preferred.

of noise resulted in higher prediction error, the mean prediction deviation for these levels is still small (less than 1.1).

4.4.2 Limited Reference Attributes. In general, we cannot predict how many reference attributes will be available. We may have very few, or we may have very many. In the process of reference attribute selection, there are two questions to consider: whether GeoAlign can make reasonable predictions with limited number of reference attributes, and how to select the reference attributes when more than one is available.

To answer these two questions, we chose multiple subsets of reference attributes among all reference attributes and repeated the cross-validated experiments for datasets in the United States. The subset of reference attributes were chosen based on their relationship with the target attribute of each test dataset. We adopted the leave- n -out metric such that $n = 1, 2$ for reference attributes with the highest (or lowest) correlation with the target attribute at the source level. The NRMSEs of these four series of experiments are compared with experiments using all reference attributes in Figure 8.

For 7 out of 10 tests, GeoAlign is making robust predictions regardless of the subset of reference attributes used. As for the series of experiments leaving 1 or 2 least target-attribute-related reference(s) out, the performance of GeoAlign is almost identical to using all reference attributes. This is in accordance with GeoAlign’s ability of assigning little weights to reference attributes loosely related to the target attribute.

Leaving out the most target-related attributes out can have an impact on accuracy. This does impact three of our attributes: area, USA uninhabited places and USPS business address datasets. None of the references are closely related to the area and the USA uninhabited places datasets at the source level (correlations less than 0.25). Apart from the two references left out, the rest of the references have even lower correlation with the target attribute (less than 0.2 and 0.05 respectively). According to the assumption basis of GeoAlign, the distribution of the target attributes is thus poorly related to the distribution of these attributes, leading to increased prediction error. We also found that leaving out the reference most related to the target attribute has almost no impact on the prediction for the USPS business address dataset; while leaving out top two such references dramatically worsens the situation. Further analysis reveals that these two references are

highly correlated with each other at the source level ($\approx 96\%$), the weight assigned to the reference most related to the target attribute is reassigned to the other when the former is left out. This verifies that similar attributes at the source level are also similarly distributed in the intersection units, as the predicted disaggregation matrix of the target attribute is almost the same regardless of using the reference most related to the target attribute or not.

These experiments give us more insight into GeoAlign reference attribute selection. GeoAlign prefers reference attributes highly related to the target attribute at the source level. For reference attributes poorly related to the target variable, it is able to weigh their contributions accordingly. The reference attributes are not necessarily independent of each other and the reference attributes are not necessarily accurate at the source level. From the user’s perspective, GeoAlign is able to make reasonable predictions by simply given all available reference attributes.

5 RELATED WORK

In the GIS community, spatial interpolation has advanced from isoline mapping in cartography to data realignment in different units or grids for multivariate analysis in geographic research [3, 31, 38]. *Realignment*, *crosswalk*, or *regridding*, is commonly used today as a preprocessing step before further data analysis in physics and socioeconomics to interpolate spatial or temporal data distribution from one grid to another [28]. Since these data are either point or areal based, two categories of methods are proposed for these two types respectively.

Areal interpolation is a subset of the spatial interpolation problem that realigns aggregates. Early methods built upon point-based interpolation, such as point-in-polygon method, do not follow the volume-preserving property such that reconstruction of exactly the original aggregates of each source unit with the transformed value of each target unit is not possible [31, 44]. It has been shown that these methods are not comparable in approximation efficiency with those that do have the property [31, 47]. Later methods thus introduce the property and turn over to the area-based areal interpolation instead [12]. These approaches depend highly on the spatial properties of the data collection area and thus different forms of ancillary data are introduced ever since.

Areal weighting method, one of the early area-based areal interpolation method, makes use of the area ancillary data available in the form of disaggregation partitions between source and target units [13, 36]. This method is widely available in GIS software for general users nowadays. However, it assumes even distribution within units (homogeneity) whereas this assumption hardly stands in reality. Areal weighting has been extended by referring to other single known reference attributes, called dasymetric weighting [1, 24, 37, 43]. These methods are restricted by the assumption of proportionality of the objective attribute to the single reference attribute. Hence the selection of the reference attribute is vital to the prediction accuracy and the methods are not adaptive to different objective attributes.

The regression methods are later introduced as extensions to the dasymetric methods allowing for multiple auxiliary variables. In general, the regression methods involve a regression of the source level data of the objective attribute on the values of the references in target units. For this track of methods, more advanced techniques such as EM algorithm, Monte Carlo simulation, smoothing techniques [9, 11, 31, 45, 46, 48], etc., are introduced later in the literature. However, they make different assumptions of density distribution within units, some of the mostly used ones are Poisson distribution and binomial distribution, and their performances are rather assumption dependent [30] and auxiliary variable dependent. Recently, more complicated regression models [35, 39, 40] are developed based on domain knowledge such as spatial correlation. However, they lack general applicability to heterogeneous target attributes and are hard to implement for practitioners.

These approaches can also be categorized as extensive or intensive approaches based on their approximation target. Extensive approaches approximate a_o^{st} while intensive ones approximate f_o^{st} . Most approaches for solving the areal interpolation problem are intensive approaches that build spatial statistical models for f_o^{st} in the disaggregation step. These approaches, mostly developed in 2-D space, can be extended to higher dimensions, though these extensions are typically non-trivial. Other major limitations of intensive approaches include narrow scope of application and low robustness to heterogeneous objective attributes.

Current intensive approaches for areal interpolation are not generally applicable for aggregate interpolation due to three main reasons. First, integration of f_o^{st} is computable in 2-D, however, it is computationally intensive in high dimensions with complex f_o^{st} . Second, shape files are indispensable for intensive approaches, and the probability density function for each intersection unit, $f_o^{st}[k]$, is associated with the shape files of source and/or intersection units. Further, attributes in plain tables without handy shape files of target units typically fail re-aggregation. Even if shape files are available, some of them constantly change over time, resulting in approximation inaccuracies. Last but not least, these approaches are not easily approachable for general users, especially those with little technical proficiency in mathematics, statistics and GIS. The \hat{f}_o^{st} model is built upon the spatial knowledge of the objective attribute; however, this knowledge is not available for all users. Further, implementations of intensive approaches are not publicly available, making them even harder to use.

Another limitation of intensive approaches is that they are not adaptive to new attributes. \hat{f}_o^{st} models are attribute dependent since the true f_o^{st} models for two attributes can be very different. Another point to note is that these approaches make many

assumptions of f_o^{st} . For instance, the distribution model of each intersection unit, the choice of parameters for these distributions and so on. Any change in these assumptions may dramatically influence the accuracy of approximation in some target unit. What is worse, there is no efficient verification of whether they are appropriate or not.

Extensive approaches are more generally applicable than the intensive ones: they can be easily extended to high dimensions, need no unit shape files, and are easy to implement. However, existing extensive approaches make use of a single reference attribute and are still limited in robustness. When the objective attribute and the reference attribute does not share similar spatial distribution, the approximated result can differ substantially from the true aggregates in target units. Further, since they use the same reference attribute irrespective of the objective attribute, they are not adaptive to different objective attributes with heterogeneous spatial distributions.

6 CONCLUSIONS AND FUTURE WORK

In this paper, we formally define the problem of aggregate interpolation in multi-dimensional space and propose GeoAlign, an adaptive multi-reference algorithm that realigns aggregates better than state-of-the-art approaches for real socioeconomic datasets. Unlike existing areal interpolation algorithms, GeoAlign requires no knowledge of spatial properties or dasymetric maps of source and target units and is thus generally applicable for plain aggregate tables. Our experiments show that GeoAlign is making better predictions in a reasonably short time. Its runtime scales linearly with the number of units in source and target levels, and is robust to noisy references even when limited references are available.

A potential future direction is to extend this work into an automatic aggregate data integration system that joins multiple aggregate tables without user intervention.

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under Grants ACI-1640575, IIS-1250880, and IIS-1743088.

REFERENCES

- [1] Branislav Bajat, Nikola Krunic, and Milan Kilibarda. 2011. Dasymetric Mapping of spatial distribution of population in Timok Region. In *Proceedings of International conference Professional practice and education in geodesy and related fields, Klavodo-Djerdap, Serbia*.
- [2] Zohra Bellahsene, Angela Bonifati, Erhard Rahm, and others. 2011. *Schema matching and Mapping*. Vol. 20. Springer.
- [3] Peter A Burrough, Rachael McDonnell, Rachael A McDonnell, and Christopher D Lloyd. 2015. *Principles of geographical information systems*. Oxford University Press.
- [4] United States Census. 2010. 2010 Census Data [Data file]. Available from <https://www.census.gov/2010census/data/>. (2010). Accessed: 2014-08-14.
- [5] NY Open Data. 2013. Facilities Licensed by the Department of Motor Vehicles (DMV License Facilities) [Data file]. Retrieved from <https://data.ny.gov/Transportation/Facilities-Licensed-by-the-Department-of-Motor-Veh/nhjr-rpi2>. (2013). Accessed: 2017-05-01.
- [6] NY Open Data. 2013. Food Service Establishment Inspections: Beginning 2005 (ACTIVE) (Food Service Inspections) [Data file]. Retrieved from <https://health.data.ny.gov/Health/Food-Service-Establishment-Inspections-Beginning-2/2hcc-shji>. (2013). Accessed: 2014-08-14.
- [7] NY Open Data. 2013. Liquor Authority Quarterly List of Active Licenses (Liquor Licenses) [Data file]. Retrieved from <https://data.ny.gov/Economic-Development/Liquor-Authority-Quarterly-List-of-Active-Licenses/hrvs-fxs2>. (2013). Accessed: 2014-08-14.
- [8] NY Open Data. 2013. NYS Attorney Registrations (Attorney Registration) [Data file]. Retrieved from <https://data.ny.gov/Transparency/NYS-Attorney-Registrations/eqw2-r5nb>. (2013). Accessed: 2017-05-01.

- [9] Arthur P Dempster, Nan M Laird, and Donald B Rubin. 1977. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the royal statistical society. Series B (methodological)* (1977), 1–38.
- [10] Cory L Eicher and Cynthia A Brewer. 2001. Dasymetric Mapping and areal interpolation: Implementation and evaluation. *Cartography and Geographic Information Science* 28, 2 (2001), 125–138.
- [11] Cory L. Eicher and Cynthia A. Brewer. 2001. Dasymetric Mapping and Areal Interpolation: Implementation and Evaluation. *Cartography and Geographic Information Science* 28, 2 (2001), 125–138.
- [12] Robin Flowerdew and Mick Green. 1993. *Developments in areal interpolation methods and GIS*. Springer Berlin Heidelberg, Berlin, Heidelberg, 73–84.
- [13] Robin Flowerdew, Mick Green, and S Fotheringham. 1994. Areal interpolation and types of data. *Spatial analysis and GIS* 121 (1994), 145.
- [14] Carl Franklin. 1992. An Introduction to Geographic Information Systems: Linking Maps to Databases. *Database* 15, 2 (April 1992), 12–21.
- [15] Esri ArcGIS Gallery. 2014. Starbucks [Map]. Retrieved from <https://services.arcgis.com/nzS0F0zdNLvs7nc8/arcgis/rest/services/Starbucks/FeatureServer>. (2014). Accessed: 2017-10-02.
- [16] Esri ArcGIS Gallery. 2017. Accidents reported to National Highway Traffic Safety Administration in 2011 (Accidents) [Map]. Retrieved from <http://services.arcgis.com/0TU5BETrBlnlhvOx/ArcGIS/rest/services/NHTSAAccidents2011/FeatureServer>. (2017). Accessed: 2017-10-02.
- [17] Esri ArcGIS Gallery. 2017. US Shopping Centers 2015 (Shopping Centers) [Map]. Retrieved from https://services1.arcgis.com/6kyLQ3wRvoPKn52L/arcgis/rest/services/US_Shopping_Centers_2015/FeatureServer. (2017). Accessed: 2017-10-02.
- [18] Esri ArcGIS Gallery. 2017. USA Cemeteries (Cemeteries) [Map]. Retrieved from <http://www.arcgis.com/home/item.html?id=5b08fa8bb5a64ea7848dc5188e47994a>. (2017). Accessed: 2017-10-02.
- [19] Esri ArcGIS Gallery. 2017. USA Public Buildings (Public Buildings) [Map]. Retrieved from <http://www.arcgis.com/home/item.html?id=d5d5b513a40145ffa60b67d9c7ab9680>. (2017). Accessed: 2017-10-02.
- [20] Esri ArcGIS Gallery. 2017. USA Uninhabited Places [Map]. Retrieved from <http://www.arcgis.com/home/item.html?id=5f0a5776cbaf4b34b9600809bf791d69>. (2017). Accessed: 2017-10-02.
- [21] Lise Getoor and Ashwin Machanavajjhala. 2012. Entity resolution: theory, practice & open challenges. *Proceedings of the VLDB Endowment* 5, 12 (2012), 2018–2019.
- [22] Michael F Goodchild, Luc Anselin, and Uwe Deichmann. 1993. A framework for the areal interpolation of socioeconomic data. *Environment and planning A* 25, 3 (1993), 383–397.
- [23] Michael F Goodchild, Nina Siu Ngan Lam, and University of Western Ontario. Dept. of Geography. 1980. *Areal interpolation: a variant of the traditional spatial problem*. London, Ont.: Department of Geography, University of Western Ontario.
- [24] I. Gregory. 2002. The accuracy of areal interpolation techniques: standardising 19th and 20th century census data to allow long-term comparisons. *Computers, Environment and Urban Systems* 26, 4 (7 2002), 293–314.
- [25] Alon Halevy, Anand Rajaraman, and Joann Ordille. 2006. Data integration: the teenage years. In *Proceedings of the 32nd international conference on Very large data bases*. VLDB Endowment, 9–16.
- [26] Mauricio A Hernández, Renée J Miller, and Laura M Haas. 2001. Clio: A semi-automatic tool for schema Mapping. *ACM SIGMOD Record* 30, 2 (2001), 607.
- [27] Esri Inc. 2017. ArcGIS Pro 2.0 [Computer Software]. Available from <https://pro.arcgis.com/en/pro-app/>. (2017).
- [28] Karen Kemp. 2008. *Encyclopedia of geographic information science*. Sage.
- [29] Hanna Köpcke and Erhard Rahm. 2010. Frameworks for entity matching: A comparison. *Data & Knowledge Engineering* 69, 2 (2010), 197–210.
- [30] Nina Siu-ngan Lam. 1982. An evaluation of areal interpolation methods. In *Proceedings, Fifth International Symposium on Computer-Assisted Cartography (AutoCarto 5)*, Vol. 2. 471–479.
- [31] Nina Siu-Ngan Lam. 1983. Spatial Interpolation Methods: A Review. *The American Cartographer* 10, 2 (1983), 129–150.
- [32] Mitchel Langford. 2006. Obtaining population estimates in non-census reporting zones: An evaluation of the 3-class dasymetric method. *Computers, environment and urban systems* 30, 2 (2006), 161–180.
- [33] Mitchel Langford, David J Maguire, and David J Unwin. 1991. The areal interpolation problem: estimating population using remote sensing in a GIS framework. *Handling geographical information: Methodology and potential applications* (1991), 55–77.
- [34] Maurizio Lenzerini. 2002. Data integration: A theoretical perspective. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 233–246.
- [35] X. H. Liu, P. C. Kyriakidis, and M. F. Goodchild. 2008. Population-density Estimation Using Regression and Area-to-point Residual Kriging. *Int. J. Geogr. Inf. Sci.* 22, 4 (Jan. 2008), 431–447.
- [36] John Markoff and Gilbert Shapiro. 1973. The Linkage of Data Describing Overlapping Geographical Units. *Historical Methods Newsletter* 7, 1 (1973), 34–46.
- [37] Jeremy Mennis and Torrin Hultgren. 2006. Intelligent Dasymetric Mapping and Its Application to Areal Interpolation. *Cartography and Geographic Information Science* 33, 3 (2006), 179–194.
- [38] Lubos Mitas and Helena Mitasova. 1999. Spatial interpolation. *Geographical information systems: principles, techniques, management and applications* 1 (1999), 481–492.
- [39] Andrew S. Mugglin, Bradley P. Carlin, and Alan E. Gelfand. 2000. Fully Model-Based Approaches for Spatially Misaligned Data. *J. Amer. Statist. Assoc.* 95, 451 (2000), 877–887.
- [40] Daisuke Murakami and Morito Tsutsumi. 2011. A new areal interpolation method based on spatial statistics. *Procedia-Social and Behavioral Sciences* 21 (2011), 230–239.
- [41] Office of Policy Development and Research (PD & R). 2010. HUD USPS Zip Code Crosswalk Files [Data file]. Available from https://www.huduser.gov/portal/datasets/usps_crosswalk.html. (2010). Accessed: 2014-08-14.
- [42] Erhard Rahm and Philip A. Bernstein. 2001. A survey of approaches to automatic schema matching. *The VLDB Journal* 10, 4 (2001), 334–350.
- [43] Michael Reibel and Michael E Bufalino. 2005. Street-Weighted Interpolation Techniques for Demographic Count Estimation in Incompatible Zone Systems. *Environment and Planning A* 37, 1 (2005), 127–139.
- [44] Yukio Sadahiro. 2000. Accuracy of Count Data Estimated by the Point-in-Polygon Method. *Geographical Analysis* 32, 1 (2000), 64–89.
- [45] Guillermo Q. Tabios and Jose D. Salas. 1985. A comparative analysis of techniques for spatial interpolation of precipitation. *JAWRA Journal of the American Water Resources Association* 21, 3 (1985), 365–380.
- [46] Waldo R Tobler. 1979. Smooth pycnophylactic interpolation for geographical regions. *J. Amer. Statist. Assoc.* 74, 367 (1979), 519–530.
- [47] Paul R Voss, David Dryden Long, and Roger Bruce Hammer. *When census geography doesn't work: Using ancillary information to improve the spatial interpolation of demographic data*.
- [48] John K Wright. 1936. A method of Mapping densities of population: With Cape Cod as an example. *Geographical Review* 26, 1 (1936), 103–110.
- [49] Yichun Xie. 1995. The overlaid network algorithms for areal interpolation problem. *Computers, environment and urban systems* 19, 4 (1995), 287–306.

Distributed query-aware quantization for high-dimensional similarity searches

Gheorghi Guzun
San Jose State University
San Jose, California
gheorghi.guzun@sjsu.edu

Guadalupe Canahuate
The University of Iowa
Iowa City, Iowa
guadalupe-canahuate@uiowa.edu

ABSTRACT

The concept of similarity is used as the basis for many data exploration and data mining tasks. Nearest Neighbor (NN) queries identify the most similar items, or in terms of distance the closest points to a query point. Similarity is traditionally characterized using a distance function between multi-dimensional feature vectors. However, when the data is high-dimensional, traditional distance functions fail to significantly distinguish between the closest and furthest points, as few dissimilar dimensions dominate the distance function. Localized similarity functions, i.e. functions that only consider dimensions close to the query, quantize each dimension independently and only compute similarity for the dimensions where the query and the points fall into the same bin. These quantizations are query-agnostic. There is potential to improve accuracy when a query-dependent quantization is used.

In this paper we propose a Query dependent Equi-Depth (QED) on-the-fly quantization method to improve high-dimensional similarity searches. The quantization is done for each dimension at query time and localized scores are generated for the closest p fraction of the points while a constant penalty is applied for the rest of the points. QED not only improves the quality of the distance metric, but also improves query time performance by filtering out non relevant data. We propose a distributed indexing and query algorithm to efficiently compute QED. Our experimental results show improvements in classification accuracy as well as query performance up to one order of magnitude faster than Manhattan-based sequential scan NN queries over datasets with hundreds of dimensions.

1 INTRODUCTION

Nearest Neighbor (NN) searches over high-dimensional data are ubiquitous in information retrieval, machine learning, and multimedia data mining. These searches are often performed through k nearest neighbor (k NN) queries over multi-dimensional feature vectors. Spatial and multimedia objects can be represented as feature vectors characterizing their shape and/or content. Social network data objects, for instance can be represented by links with other data objects, history actions, preferences, etc. Application domains such as spatial data-bases[8], computer vision [5], multimedia and social network applications [29] can all benefit from a more efficient method for finding nearest neighbors in high dimensional spaces.

However, due to the rapid advancements in data generation and collection, it is increasingly challenging to process similarity searches in a rapid and meaningful way on these larger and more complex datasets. Existing methods for finding nearest neighbors

using tree-based indexing for spatial data [36] and low dimensional data [13, 23, 24], suffer from the curse of dimensionality when applied to high-dimensional spaces. Moreover, not only processing time degrades, but also the ability to characterize similarity using a distance function for high-dimensional spaces is greatly reduced[3]. The reason is that distances between data points in high-dimensional spaces, are usually very concentrated around their average [7]. This makes it difficult to distinguish between the closest and furthest data points [3].

To overcome this limitation, localized distance functions [1, 37, 39] have been proposed in the literature. These functions only consider dimensions that are close to the query point to characterize similarity providing a better distinction between closer and further points and often improving the accuracy of the results. The IGrid index [1] efficiently supports the computation of a partial distance called PiDist and its performance scales well for high dimensional data. IGrid pre-process the data and define equi-populated partitions (bins) over each dimension independently. The points in these partitions are then mapped to buckets and only the points that fall into the same bucket as the query point are considered similar for that dimension. The points with the largest cumulative similarity are then retrieved after all dimensions have been processed. As stated in the paper, the accuracy of the results are affected by the binning strategy and the number of bins used.

In this paper, we expand on the ideas proposed in [1, 15] and define a query-dependent quantization strategy that further improves the accuracy of the results. The main idea is to use the query itself to determine a range for which the points falling within are considered similar. Note that considering a fixed interval around the query value for each dimension is not a viable option. Since the distribution of each attribute varies, a small interval could yield empty bins, while a large interval could include all the points. Determining the range needs to consider the data values in the attribute. We want to define a bin for each dimension where the number of points is roughly the same for each dimension. This novel function, called Query dependent Equi-Depth (QED) quantization, can be used with traditional distance functions (e.g. Euclidean, Manhattan) to improve their accuracy in high-dimensional spaces. In order to efficiently support QED over big-data, we design a distributed indexing and efficient query algorithm. The proposed approach includes the usage of compressed bitmap-based indexing and low level parallel bitwise operations.

Our approach does not require any pre-computations for quantization, or excessive storage or memory. On the contrary, the index requires less storage than the data itself. The index and query algorithms are designed for parallelism and can run on centralized systems as well as cluster systems. In this work we evaluate the distributed indexing and query processing on a Spark cluster, however it is suitable for other distributed environments as well. With QED quantization, as shown in our performance evaluation we observe an improvement in query time of up to one

order of magnitude when compared to Sequential Scan on high dimensional data.

We also evaluated the kNN classification accuracy of the proposed QED quantization on a set of nine high-dimensional datasets and observed an average improvement in accuracy of 2.4% for Manhattan distance and 10.95% for Hamming distance when using QED quantization.

The primary contributions of this paper can be summarized as follows:

- We present a bitmap-based, distributed index for answering similarity searches and nearest neighbor queries. The cumulative distance is computed in parallel through an optimized distributed aggregation that uses task mapping by slice depth at its core.
- We formalize the cost for the distributed aggregation and optimize the partition size to balance the memory shuffling and parallel processing trade-off.
- We propose a novel Query dependent Equi-Depth quantization (QED) for improved similarity searches. This dynamic quantization not only improves accuracy but also execution time because it reduces the amount of data processed by only considering, for each dimension, the closest points to the query.
- We introduce a power function using the number of tuples and the number of dimensions as a heuristic to determine the number of points that are considered similar in each dimension and empirically evaluated it.
- We evaluate the kNN classification accuracy of QED on a number of labeled real datasets and the performance of the proposed index and query algorithms in a distributed setting.

The rest of the paper is organized as follows. Section 2 presents related work on NN searches. Section 3 describes the index structure, the proposed quantization, and the parallel query optimization for similarity searches. Section 4 provides an experimental evaluation of QED over a Spark/Hadoop cluster. Finally, conclusions are presented in Section 5.

2 RELATED WORK

This section presents related work for high-dimensional similarity searches and nearest neighbor queries. The most recent exact nearest neighbor searches are using localized similarity functions. While some of the most recent approximate nearest neighbor searches use hashing or product-quantization techniques.

2.1 High-dimensional Similarity using Localized Functions

As described in [37], for high-dimensional applications where human cognition is the target judge of object similarity, it is more important to closely match a subset of attributes rather than provide some least total distance measurement over all the attributes. The similarity function, called Dynamic Partial Function (DPF) [37], considers only the smallest N distances between all dimensions to compute the similarity. Since the method is so sensitive to N , the k-N-match problem is introduced in [37] and the authors propose the use of a frequent k-N-match algorithm, where the most frequent k objects appearing in the k-N-match solutions for a range of N values. It is worth noting that DPF is not a distance since the triangle inequality does not hold.

In [1], quantization was used to create equi-populated partitions for each dimension to bound the worst case performance

of the query. Quantization has been widely used to improve the accuracy of classifiers and clustering algorithms as it reduces the noise of the data and simplifies the models. Supervised methods use the class label and a training dataset to make an informed decision about the optimal split points. Unsupervised methods rely solely on the statistics collected about the data. Examples of unsupervised methods are equi-width (divide into intervals of the same length) and equi-populated or equi-depth (divide into intervals with the same number of data objects). Examples of supervised methods are entropy-Minimum Description Length Principle (MDLP) [9], chi-merge [25], class-attribute interdependent maximization (CAIM) [26], and Class-attribute Contingency Coefficient (CACC) [38], among others. A detail survey of quantization methods can be found in [11].

After quantization, each attribute is represented using discrete values and points are considered “close” if they fall into the same bin. Hamming distance is the preferred distance metric for discrete domains and is defined as:

$$\text{Hamm}(x, y) = \sum_{i=1}^d \begin{cases} 0 & \text{if } x_i = y_i \\ 1 & \text{otherwise} \end{cases}$$

where x and y are the high-dimensional quantized vectors, d is the number of dimensions and x_i denotes the value of x for dimension i . The evaluation of the Hamming function produces discrete values in the range $[0, d]$. This is a source of ambiguity when defining similarity as the distance score is the same for an increasing number of nearest neighbors. To break these ties a weighted hamming distance function can be used. `PiDist` [1] computes the normalized Manhattan distance over the continuous values for the dimensions where the two points fall into the same discretization range. When equi-populated ranges are used, the attribute is forced to follow a uniform distribution and the distance between the continuous values is used to capture similarity.

`PiDist` is defined as:

$$\text{PiDist}(X, Y, k_d) = \left[\sum_{i \in S[X, Y, k_d]} \left(1 - \frac{|x_i - y_i|}{m_i - n_i} \right)^p \right]^{\frac{1}{p}}$$

where k_d is the number of splits for each dimension, $S[X, Y, k_d]$ is the set of dimensions for which the two objects lie in the same range, and m_i and n_i are the upper and lower bounds of the corresponding range in dimension i .

This function accumulates benefit for each attribute for which a data object maps to the same quantization as the query object. It does not differentiate between data and query objects that do not map to the same quantization. Therefore, a data point is not excessively penalized for a few dissimilar attributes.

2.2 Approximate nearest neighbor searches

Given the difficulty of answering exact nearest neighbor searches, approximate nearest neighbor searches were introduced to solve the NN problem in high dimensional spaces [10, 12, 20–22]. Local Sensitive Hashing (LSH)[12] hashes the data so that similar items fall into the same buckets. However, data sets are typically not distributed uniformly over the space, and as a result, the buckets of LSH are unbalanced, causing the performance of LSH to degrade. Data Sensitive Hashing (DSH) [10] aims to keep the buckets balanced with the help of a new hash family, while preserving the nearest neighbor relations.

Hashing techniques require the pre-computation of the hash tables without prior knowledge of the query. The accuracy of a similarity search is determined by the number and the quality of

Tuple	Raw Data		Bit-Sliced Index (BSI)				BSI SUM		
	Attrib 1	Attrib 2	Attrib 1		Attrib 2		sum[2]	sum[1]	sum[0]
			$B_1[1]$	$B_1[0]$	$B_2[1]$	$B_2[0]$			
t_1	1	3	0	1	1	1	1	0	0
t_2	2	1	1	0	0	1	0	1	1
t_3	1	1	0	1	0	1	0	1	0
t_4	3	3	1	1	1	1	1	1	0
t_5	2	2	1	0	1	0	1	0	0
t_6	3	1	1	1	0	1	1	0	0

Figure 1: Simple BSI example for a table with two attributes and three values per attribute.

the hash functions along with other tuning parameters. A higher number of hash functions can result in higher a probability of grouping similar objects together, however this can also result in a significant storage overhead. Moreover, with addition of new data, the hash index has to be re-computed. While QED uses some of the same concepts by projecting data into smaller ranges, it does not come with the storage overhead required by the hash index, and uses the query directly for data projection. Moreover, QED is an exact similarity function.

We compare QED against a distributed LSH¹ implementation and show that QED could present advantages in terms of smaller index size, and better accuracy for some applications.

3 DISTRIBUTED QED FOR HIGH DIMENSIONAL SIMILARITY SEARCHES

Consider a relation R and query vector Q . Every object in R is represented by m attributes or numeric scores. The query vector $Q = \{q_1, \dots, q_m\}$ is also represented by m values. The task is to find the k most similar objects to Q in R . This k nearest neighbor (k NN) query can either compute the similarity between each data object and the query and retrieve the k objects with the highest score or, inversely, compute the distance between each object and the query and retrieve the k objects with the smallest distance.

In high dimensional spaces however, many distance functions such as Manhattan and Euclidean metrics are not as effective and the quality of the answer returned by the k NN query degrades. The reason these functions are directly affected by the dimensionality of the data lies in the fact that the dominant components are the dimensions for which two points are farthest apart. With higher dimensionality, the probability of having high discrepancies between two points in at least one dimension increases. The authors of [1, 3] show that for L_p -norm distance functions, the averaging effects of the different dimensions start predominating with increasing dimensionality. To prevent this, the authors of PiDist [1] show that imposing a proximity threshold for each dimension, beyond which the degree of dissimilarity is not relevant, could improve accuracy in nearest neighbor and similarity searches.

They achieve this by quantizing the indexed space into a fixed number of bins which are either equi-width or equi-depth (equi-populated). These quantizations are performed over the dataset without considering the query points. Even when a query point lies close to the boundary of a bin, only the points within the bin are considered for computing the similarity. In this section we describe a novel equi-depth quantization method that considers the query value for defining the bin boundaries and it is done on-the-fly during query execution.

3.1 Bit-sliced indexing

In order to efficiently support QED over large datasets, we design a distributed indexing and efficient query algorithm. The proposed approach includes the usage of compressed bit-vector indexing and low level parallel bitwise operations. As shown previously [16, 18], this setup can leverage SIMD instructions and use the processing hardware more efficiently, for arithmetic operations.

Bit-sliced indexing (BSI) was introduced in [30], and it encodes the binary representation of attribute values with binary vectors. Therefore, $\lceil \log_2 values \rceil$ vectors, each with a number of bits equal to the number of records, are required to represent all the values for a given attribute.

Figure 1 illustrates how indexing of two attribute values and their sum is achieved using bit-wise operations. Since each attribute has three possible values, the number of bit-slices for each BSI is 2. For the sum of the two attributes, the maximum value is 6, and the number of bit-slices is $\lceil \log_2 6 \rceil = 3$. The first tuple t_1 has the value 1 for attribute 1, therefore only the bit-slice corresponding to the least significant bit, $B_1[0]$ is set. For attribute 2, since the value is 3, the bit is set in both BSIs. For example, the addition of the BSIs representing the two attributes is done using efficient bit-wise operations. First, the bit-slice $sum[0]$ is obtained by XORing $B_1[0]$ and $B_2[0]$: $sum[0] = B_1[0] \oplus B_2[0]$. Then $sum[1]$ is obtained in the following way: $sum[1] = B_1[1] \oplus B_2[1] \oplus (B_1[0] \wedge B_2[0])$. Finally $sum[2]$, which is the carry, is $majority(B_1[1], B_2[1], (B_1[0] \wedge B_2[0]))$, where $majority(A, B, C) = (A \wedge B) \vee (A \wedge C) \vee (B \wedge C)$.

BSI arithmetic for a number of operations, including the addition of two BSIs, is defined in [34]. Previous work [19, 33], uses BSIs to support preference and top k queries efficiently. BSI-based top k for high-dimensional data [16, 19] was shown to outperform current approaches for centralized and distributed query processing. In this work we adapt the distributed BSI query processing for NN queries. Furthermore, each individual bit-vector is compressed. The compression mechanism is described in section 3.6.

3.2 Query Dependent Equi-Depth Quantization

Further in this section we describe a novel function, called Query dependent Equi-Depth (QED) quantization, which can be used with traditional distance functions (e.g. Euclidean or Manhattan) to improve their accuracy in high-dimensional spaces. QED is implemented using a distributed bitmap-based index, and is designed with performance considerations in mind.

The main idea with localized functions is that for each dimension, if the data point has its respective dimension within threshold x then the distance to the query is considered for that dimension otherwise a dissimilarity penalty larger than x is assigned.

For this work, instead of directly specifying x , we consider parameter p as the minimum number of data points that should be contained within the query bin boundaries. Parameter p is

¹<https://github.com/mrsqueeze/spark-hash>

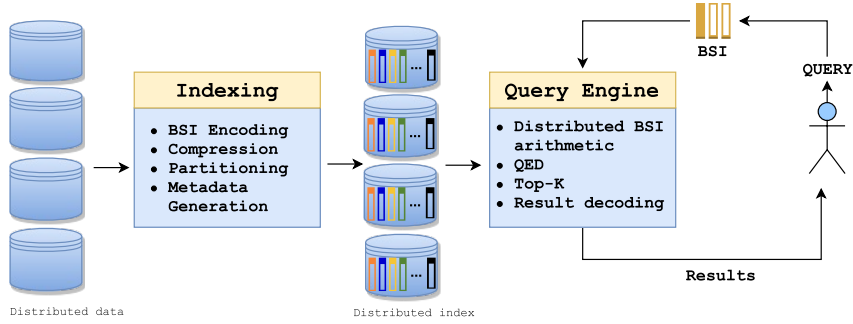


Figure 2: High level system overview

expressed as a percentage. Given the query value for dimension i , q_i , we find the $\lceil pn \rceil$ smallest distances to q_i and define a similar bin around the query point.

To illustrate the proposed dynamic quantization, consider a 1-dimensional dataset with values:

$$\{\{r1, 9\}, \{r2, 2\}, \{r3, 15\}, \{r4, 10\}, \\ \{r5, 36\}, \{r6, 8\}, \{r7, 6\}, \{r8, 18\}\}$$

and query $\{q, 10\}$. If using Manhattan distance, the distance between the data points and the query are:

$$\{\{r1, 1\}, \{r2, 8\}, \{r3, 5\}, \{r4, 0\}, \\ \{r5, 26\}, \{r6, 2\}, \{r7, 4\}, \{r8, 8\}\}$$

For QED, if parameter $p = 0.35$ (35% of the population), only the 3 points with the smallest distances, i.e. $\{r1, r4, \text{ and } r6\}$, will be considered according to their distance. The rest of the points will be given a larger penalty δ_i to characterize a large dissimilarity. This normalization of the larger differences gives point $r5$ a chance to make it as a NN in the cases where there are other many dimensions for which $r5$ is really close to the query.

The value of the penalty, δ_i , can be assigned a constant larger than the distances computed within the query-dependent interval for each dimension. In the case of PiDist, the penalty assigned to dissimilar points is 1 and the distance for similar points is normalized to less than 1. Another approach could be to make δ_i to represent a number larger than the largest distance between the query and the closest p elements in dimension i .

Equation 1 shows the Manhattan distance between a data point a and the query q after applying the QED quantization.

$$\text{QED}_{\text{Manhattan}}(a, q) = \sum_{i=1}^d \begin{cases} |a_i - q_i| & \text{if } a \in P_i \\ \delta_i & \text{otherwise} \end{cases} \quad (1)$$

Where P_i is the subset of points closest to the query in dimension i , and δ_i is the penalty for the points outside the similar range in dimension i .

Performing QED for kNN queries without an index would slow the execution time, as it needs to dynamically compute a range for each dimension based on the query, in addition to computing the distance. We choose to implement QED on top of a bit-sliced index because BSI provides a compact representation of numeric values and an implicit ordering of the values as the set bits in the most significant bit-slice represent the largest values. In the next section we show how using the BSI index can in fact improve the performance of the kNN query.

3.3 Distributed Bit-Sliced Index (BSI)

Figure 2 depicts a high level overview of the proposed system. There are two main components: the indexing module and the query engine. The indexing module encodes each attribute into a bit-sliced index (BSI), compresses and partitions them, and generates the metadata required by the query engine to ensure correctness of the execution plan. The query engine encodes the user query into a BSI, runs the distributed kNN query, and returns the k nearest neighbors to the query.

We support both vertical (a subset of the attributes) and horizontal (a subset of the rows) partitioning. Distributed query algorithms are developed to minimize shuffling, improve load balancing, and maximize cluster utilization. For this work, we use Apache Spark and its Java API to distribute the workload across the cluster.

3.3.1 Indexing. Let us denote by B_i the bit-sliced index (BSI) over attribute i . $B_i[j]$ is a binary vector containing n bits (one for each tuple), where j represents the j^{th} bit in the binary representation of the attribute value. j can hold a value between 0 and the number of slices s used to represent values from 0 to $2^s - 1$. The bits are packed into words, and each binary vector encodes $\lceil n/w \rceil$ words, where w is the computer architecture word size (64 bits in our implementation).

For every attribute i in R we create a Bit-sliced index B_i . The BSI index is then partitioned and distributed across the nodes of a cluster. The $BSIAttr$ class serves as a data structure for an atomic BSI element included in a partition. Each partition can include one or more $BSIAttr$ objects. A $BSIAttr$ object can represent all the attribute tuples (in the case of vertical-only partitioning) or only a subset (in the case of horizontal, or vertical and horizontal partitioning). Furthermore, a $BSIAttr$ object can carry all of the attribute bit-slices or only a subset of them. The $BSIAttr$ metadata generated includes information regarding the data type, encoding, number of slices, partition mapping. An example of attribute partitioning and use of metadata for partition mapping is shown in Figure 3.

We extended the BSI to handle signed numbers (both 2's complement and sign and magnitude) and represent decimal numbers using a fixed point format for each attribute. For every decimal BSI, the position of the decimal point is maintained as metadata for the attribute. To perform arithmetic operations between two attributes with different precision, namely a and b , where $a > b$, the decimal point for the second attribute is moved $(a - b)$ positions by multiplying the second attribute by the appropriate power of 10. Multiplication by a constant, as in this case, can be done efficiently by adding the logically shifted BSI to the original BSI for every set bit in the binary representation of the constant.

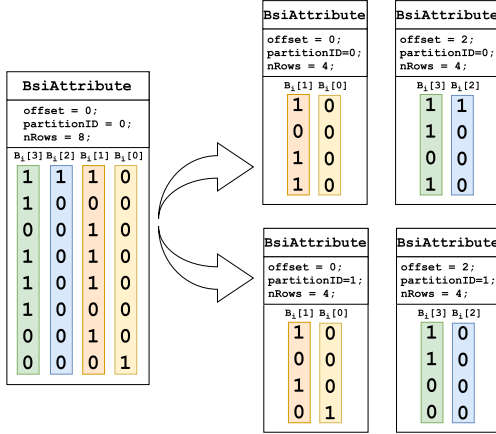


Figure 3: Example of vertical and horizontal partitioning of a BSI Attribute.

At query time, a BSI index is also generated for each partition using the attribute values in the query. Since the query value is constant, compressed bit-slices of all 0s or all 1s are used to generate a BSI with as many bits as objects in the partition in order support the bit-wise operations between the query and the *BSIAttr*. The hybrid query execution model [14] allows us to operate compressed and verbatim bit-vectors together and the results are dynamically compressed/decompressed as needed.

3.3.2 Query Engine. For developing the distributed algorithm we identified three main steps in the *k*NN query processing: compute the distance between the query *Q* and each *BSIAttr*, accumulate the partial distances for all dimensions, and retrieve the *k* closest points as the answer to the query. For an efficient execution of these three steps we are proposing a dynamic query aware quantization method called QED, and integrate it with algorithms for parallel execution of BSI arithmetic operations. In the following sections we describe in more detail the different components of the query engine.

3.4 Distributed *k*NN Query Processing

Most parts of the three-step process described earlier must be executed in parallel for achieving good scalability and performance. The computation of the absolute value of the difference between the query and each attribute BSI for *all* dimensions in parallel, aggregating all the distances into a single SUM_BSI also in parallel, and performing the top-*k* operation over the result BSI (can be executed in a single node or in parallel). We apply the same slice mapping distributed BSI aggregation developed for preferences queries [16]. This approach, described next, outperforms other parallel baseline implementations such as tree-reduction (adding pairs of BSIs together and using multiple reduce rounds) and its optimization Group Tree Reduction that reduces together groups of BSIs to reduce the number of rounds and the amount of data shuffled.

3.4.1 SUM_BSI Using Slice Mapping. It is true that the compact representation of the BSI makes the baseline implementations highly competitive versus their array counterparts. However, most of the performance gains, if not all, come from the reduced size of the BSI, and not necessarily because the algorithms are efficient. We propose an aggregation algorithm that promotes the bit-slices as the processing data units and applies the lessons-learned in computer arithmetic optimization to further improve

the performance of the parallel aggregation. The basic idea of this approach lies in the use of the bit-slice depth as the mapped key and implement a two-phase aggregation algorithm, shown in Figure 4. In the first phase, the slices are added by bit-depth, producing a weighted partial sum BSI. In the second phase, all the partial sums are added together in a method similar to a carry-save adder.

Consider a dataset where $m = 128$ attributes are added using 10-nodes. Let us now assume that each attribute’s value is within $1M = 2^{20}$, so every attribute i can be further partitioned into a set of 20 vertical bit-slices: $\{B_i[d] \mid 19 \geq d \geq 0\}$. In the proposed two-phase algorithm, the first task is to map all the bit-slices with the same depth (d) to a single node. Then addition is performed over 128 BSIs containing only 1 slice each, producing 20 partial sum BSIs. Each partial sum is in the range $[0, 128]$ and would require at most 8 slices. Next, these partial sums are added using their original depth d as their “weight.” For example, the partial sum for the bit-slices of depth $d = 2$ would have a weight of $2^d = 4$. Because the weight is always a power of 2, this weighting scheme can be done efficiently by bit-shifting. Since the BSIs are stored column-wise, this shift can be represented using an offset and never materialized.

It is also possible to perform the parallel aggregation using groups of bit-slices to reduce data shuffling. In the previous example, with a group size of $g = 2$, we could have slices 0 and 1 from all 128 attributes added together in the same node during the first stage. This ability to group the slices and divide the attributes (e.g., half of the depth 0 slices added in one node and the other half in another), allows us to balance the load and keep all the nodes busy longer.

For clarity in describing our algorithms, we use the example illustrated in Figure 4. In the first phase, every BSI attribute has its slices mapped locally to based on their depth d . The splitting of the BSI attribute in individual bit-slices allows for a finer granularity of the indexed data and for a more efficient parallelism during the aggregation phase. The pseudo-code of the mapping step is shown in the first Map() function of Algorithm 1. Every mapper has a *BSIAttr* (containing multiple slices) as input, and outputs a set of *BSIAttrs* that contain *one* bit-slice each. These bit slices are mapped by their depth in the input *BSIAttr*. Although there is an overhead associated with encapsulating each bit-slice into a *BSIAttr*, by creating a higher level of parallelism, we also achieve better load balancing and resource utilization.

Still in the first phase, the aggregation is done by the Reduce-ByKey() function of Algorithm 1. In this step, all the bit-slices with the same key (depth) are aggregated into a *BSIAttr*. Line 9 of Algorithm 1 performs the summation of two BSIs. We use the same addition logic as the authors in [35]. However, we achieve a parallelization of the BSI summation algorithm by splitting the *BSIAttr* into individual slices and executing their addition in parallel similarly to a carry-save adder. The offset of the resulting *BSIAttr*s are saved in the *offset* field of each *BSIAttr* object to ensure the correctness of the final aggregated result. The summation is optimized by aggregating the bit-slices on the same node first, then on the same rack, and then across the network. Thus, trying to minimize the network throughput. The aggregation by depth is done locally first.

After aggregating partial local results, the second phase initiates to complete the aggregation by depth through shuffling the partial sums and reducing by their depths. The final step of the aggregation is done by reducing all the BSIs (*pSum*) produced in

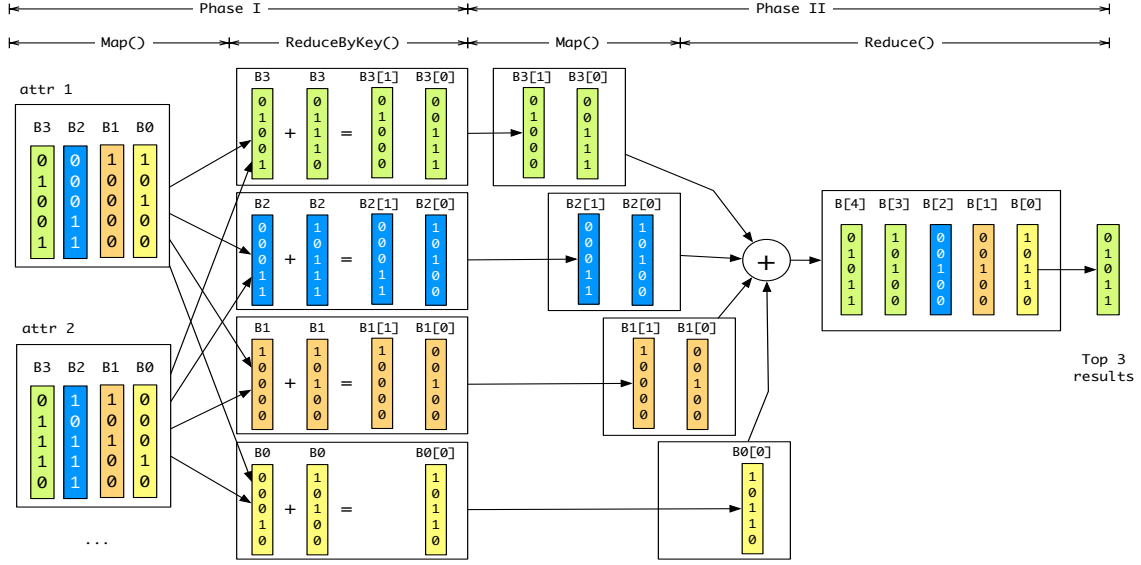


Figure 4: SUM_BSI Using Slice Mapping Example

the previous ReduceByKey() stage, regardless of their key. The final result ($attSum$) of this reduce phase is a single BSI attribute in the case of vertical only partitioning, or a set of BSI attributes, that should be concatenated, in the case of vertical and horizontal partitioning. Concatenation is straight forward, as each BSI in a partition has the same number of bits corresponding to the same rowIds.

3.4.2 Cost Estimations and Query Optimization. It is possible to estimate the complexity of the two-phase distributed aggregation and the expected amount of data shuffling. These estimations can help in choosing the most optimal partitioning strategies (group slicing) for the distributed aggregation, thus finding the best compromise between parallelism and the cost of network communication.

Note that the mapping in the first phase (Figure 4) does not produce any shuffling since it aggregates only the slices from attributes found on the same node. Data shuffling occurs twice in our two-phase aggregation. The first time is between the reducers of phase 1 and the mappers of the phase 2, and the second time data is shuffled between the mappers and reducers of the second phase. The amount of data shuffled depends on the number of nodes, partitions, tasks (or the number of attributes per task), and the number of slices per group. The number of slices per group can vary from 1 to s , where s is the highest number of slices per attribute in the dataset. In Figure 4 the slices are mapped into groups of one.

In order to determine the amount of data shuffled between the phase 1 and the phase 2, we should find first the number of outputs created by the reducers of phase 1. Given m attributes with s maximum slices per attribute, a attributes per node, and g slices per group, each node produces $\frac{s}{g}$ partial aggregations by depth. The size of each of these partial aggregations is in the worst case:

$$\lceil \log_2(g + a) \rceil \quad (2)$$

This represents the number of slices each partial aggregation by depth contains after the reduce phase 1. The total number of

slices shuffled at this stage is:

$$Sh_1 = \left[\left(\min \left(\left\lceil \frac{s}{g} \right\rceil, \left\lceil \frac{m}{a} \right\rceil \right) - 1 \right) \cdot \left\lceil \frac{m}{a} \right\rceil \cdot \lceil \log_2(g + a) \rceil \right] \quad (3)$$

Algorithm 1: Two phase distributed BSI aggregation by slice depth

```

Map(): //Map slices by depth
begin
  Input: RDD<BSIAttr> indexAtt
  Output: RDD<Integer, BSIAttr> byDepth
  int sliceDepth=0;
  while indexAtt has more slices do
    bsi = new BSIAttr();
    bsi.add(indexAtt.nextSlice());
    byDepth.add(new Tuple(sliceDepth, bsi));
    sliceDepth++;
  end
  return byDepth
end
ReduceByKey()://Reduce by depth - first reduce phase
begin
  Input: RDD<Integer, BSIAttr> byDepth1, byDepth2
  Output: RDD<Integer, BSIAttr> pSum
  pSum = byDepth1.SUM-BSI(byDepth2);
  return pSum
end
Map():
begin
  Input: RDD<Integer, BSIAttr> partSum
  Output: RDD<BSIAttr> pSum
  pSum = partSum._2();
  return pSum
end
Reduce(): //Second reduce phase
begin
  Input: RDD<BSIAttr> pSum1, pSum2
  Output: RDD<BSIAttr> sumAtt
  sumAtt = pSum1.SUM-BSI(pSum2);
  return sumAtt
end

```

The mappers of the second phase produce $\frac{s}{g}$ outputs, each with the size:

$$\lceil \log_2(g+a) \rceil + \lceil \log_2\left(\frac{m}{a}\right) \rceil = \lceil \log_2\left(\frac{(g+a)m}{a}\right) \rceil \quad (4)$$

The total number of slices shuffled between the mappers and reducers of the second phase is:

$$Sh_2 = \left(\left\lceil \frac{s}{g} \right\rceil - 1 \right) \lceil \log_2\left(\frac{(g+a)m}{a}\right) \rceil \quad (5)$$

The total amount of data shuffled is the sum of the results from Equations 3 and 5:

$$Sh = Sh_1 + Sh_2. \quad (6)$$

The amount of data shuffled decreases as g - the number of slices per group increases, or as a - the number of attributes per node increases. However, less data shuffling means a higher load on individual tasks. We further analyze the time complexity for each individual task, and its impact on the total query time in the two-phase distributed aggregation.

The cost of summing two BSI attributes is linear on the number of slices and the number of rows in the attributes. If v is the number of slices of the attribute with a higher number of slices, then the cost of adding the two attributes is equal to the cost of executing v bitwise logical operations between two vectors. Given that the number of slices per group is a constant, g is the number of slices for each depth-shifted attribute in the reduce phase 1. Adding all the depth-shifted attributes within one node has the following complexity:

$$T_1 = \sum_{i=1}^{\log_2 a} (g+i). \quad (7)$$

There are $\frac{m}{a}$ partial sums with the same key per task, to complete the aggregation of partial sums shifted by depth. Thus the cost of this aggregation is:

$$T_2 = \sum_{i=1}^{\lceil \log_2 m/a \rceil} (g + \lceil \log_2 a \rceil + i) \quad (8)$$

Finally, the cost of aggregating the partial sums shifted by depth into one final attribute, is given by:

$$T_3 = \sum_{i=1}^{\lceil \log_2 s/g \rceil} \left(g + \lceil \log_2 a \rceil + \lceil \log_2 \frac{m}{a} \rceil + i \right) \quad (9)$$

When taking into consideration the time complexities from Equations 7, 8, and 9, one must account for the different number of tasks executed in these three steps. For example, if T_1 has a weight of one, i.e. $W_{T_1} = 1$, then the number of tasks for T_2 and T_3 is different. For $W_{T_1} = 1$, the weight for T_2 is:

$$W_{T_2} = \frac{1}{\lceil \frac{m}{a} \rceil} \quad (10)$$

since there are fewer tasks for T_2 than T_1 by a factor of $\frac{m}{a}$. While the weight for T_3 is:

$$W_{T_3} = \frac{1}{\lceil \frac{m}{a} \rceil \lceil \frac{s}{g} \rceil} \quad (11)$$

In this case, there are s/g fewer tasks than in the previous step.

Using the time complexities discussed above, together with the data shuffle estimations, it is possible to find the optimum values for the number of slices per group (g) and the number of initial tasks/attributes per task.

3.5 QED over the distributed BSI

QED can be done gracefully with the BSI index without imposing any overhead when compared to the computation of the Manhattan distance without indexing, as shown in Algorithm 2. We operate on top of a BSI index representing the distance between the query and the data points in each dimension. Thus we define the penalty δ_i as the truncation of the most significant bits for the largest distances as depicted in Figure 5.

Algorithm 2: QED Quantization

Input: BSI A , int p
Output: BSI S

```

1 BitSlice penalty = ( $A[A.size - 2]$  XOR  $A.sign$ );
2 for ( $i = A.size - 2; i \geq 0; i--$ ) do
3   | penalty = (penalty OR ( $A[sSize]$  XOR  $A.sign$ ));
4   | if penalty.count()  $\geq n - p$  then
5     |   | sSize =  $i$ ;
6     |   | break;
7   | end
8 end
9 BSI  $S =$  new BSI(sSize);
10 for ( $i = 0; i < sSize; i++$ ) do
11   |  $S[i] = (A[i]$  XOR  $A.sign)$ ;
12 end
13 S.addSlice(penalty);
14 return  $S$ 
```

QED can be included in the calculation of the absolute value of the distance between query and each dimension as shown in Algorithm 2. In datasets with large attribute ranges, the output of Algorithm 2 is significantly smaller in size than the size of the actual distance measures, for most distributions. This is very important because the result of this operation is further processed to aggregate and rank similar objects. As a result of reducing significantly the output size of this step, the overall execution time of the k NN query is generally improved. The number of bit-vectors required to encode a difference attribute is equal to the number of bits required to encode the difference range. Where the difference range is the maximum difference between the query dimension and the same dimension of any of the $[pn]$ tuples, and their minimum difference.

In large datasets where the number of tuples is high, p should typically be small, and most of those p closest tuples are much closer than the attribute range. Thus the reduction in size of the result of Algorithm 2. A more detailed discussion on parameter p follows in section 3.5.1.

For a better understanding of Algorithm 2, we show how the distance BSI attribute between the query and the data points used in our previous running example is quantized using QED in Figure 5. For simplicity, all the distances are positive in Figure 5 (leftmost BsiAttribute). Starting from the most significant bit-slice, the bit-slices in the distance attribute are OR-ed until the count of set-bits in the resulting bit-slice (the penalty bit-slice) is equal or greater than $(n - p)$. At this point the bit-slices that were operated are dropped and replaced with one single penalty bit-slice.

The effect of this quantization is the identification of the furthest $(n - p)$ points from the query for one given dimension, and reducing their distance, while keeping an accurate distance for the close points. Hence avoiding over penalizing a point if only a few dimensions are far from the query.

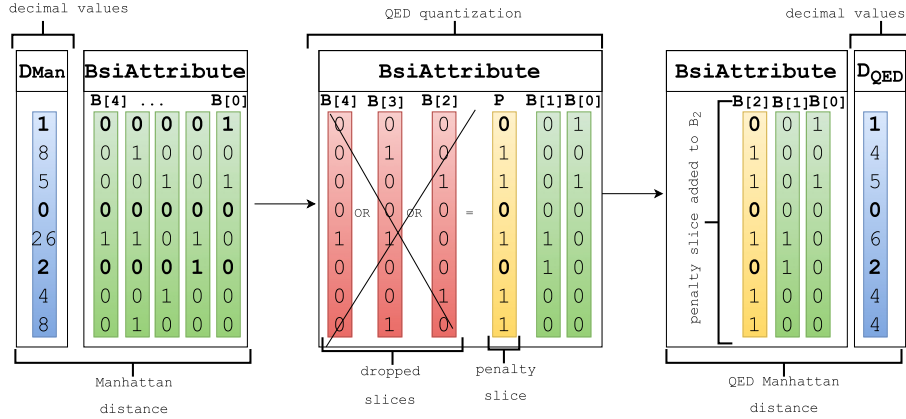


Figure 5: Query dependent Equi-Depth (QED) quantization with population range $p = 35\%$

Figure 5 and Algorithm 2 use Manhattan distance along with QED quantization. However it is also possible to use other distance metrics such as Euclidean or Hamming.

Equation 12 shows the Hamming distance between a data point a and the query q after applying the QED quantization.

$$\text{QED}_{\text{Hamming}}(a, q) = \sum_{i=1}^d \begin{cases} 0 & \text{if } a \in P_i \\ 1 & \text{otherwise} \end{cases} \quad (12)$$

Where P_i is the subset of points closest to the query in dimension i .

3.5.1 Estimating parameter p . The main idea of QED is to use the query itself for determining a range in which the points falling within are considered similar. Determining this range needs to consider the data values in the attribute. Thus we want to define a bin for each dimension where the percentage of points p in each bin is roughly the same for each dimension. We define p as a fraction of the total number of rows n in the dataset.

The value of p is directly influenced by the data dimensionality and the total number of rows in the dataset. Intuitively, for large datasets with a large number of tuples, p should be small, as even a small p would represent a large number of candidate points. Conversely, as the number of dimensions increases, p should also increase to prevent all the tuples from being penalized in many dimensions.

Inspired by the Pareto principle [31], where only the vital few produce the majority of results, we define the power function given in Equation 13 as a heuristic to estimate p :

$$\hat{p} = \left(\frac{m}{m+n} \right)^{\frac{1}{\lg(n)}} \quad (13)$$

For this power distribution, we use the number of attributes, m , as the scale, and the number of tuples, n , to derive the shape. We made the power function $\frac{m}{m+n}$ to guarantee a number less than 1.

Figure 6 shows the estimated values of p for four datasets with 1M, 10M, 100M, and 1B tuples as the number of attributes increases. We empirically evaluated this estimations over two large datasets and observed that the estimations for p were at or near the point with maximum accuracy for the task of k NN classification.

3.6 Bitmap Compression

For further optimization, in our setup, we apply compression to each individual bit-vector, when suitable [14]. Most types of

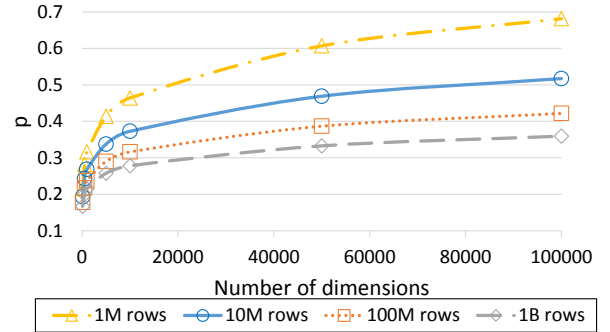


Figure 6: Estimated values of parameter p for maximizing accuracy of the k NN query results with QED

bitmap (bit-vector) compression schemes use specialized run-length encoding schemes that allow queries to be executed without requiring explicit decompression.

Word-Aligned Hybrid Code (WAH) [40] proposes the use of words to match the computer architecture and make access to the bitmaps more CPU-friendly. WAH divides the bitmap into groups of length $w - 1$, where w is the CPU's word size. WAH then collapse consecutive all-zeros or all-ones groups into a fill word.

Recently, several bitmap compression techniques that improve on WAH by making better use of the fill word bits have been proposed in the literature [27, 41], and others. Previous work have also used varying segment lengths s within $s \leq w$ encoding [17].

In this work we use our recently proposed bit-vector compression scheme [14], which is a hybrid between the verbatim scheme and the EWAH/WBC [27] bitmap compression. This hybrid scheme compresses the bit-vectors if the bit density is below a user-set threshold. Otherwise the bit-vectors are left verbatim. In our experiments we begin with compressed bit-vectors if the compressed size for the bit-vector is 0.5 or smaller than the size of the uncompressed bit-vector. The query optimizer described in [14] is able to decide at run time when to compress or decompress a bit-vector, in order to achieve faster queries. We choose this compression scheme due to its capability of operating with denser bitmaps, which is the case for the bit-vectors inside the bit-sliced index, and it allows for uncompressed bit-vectors to be operated with compressed ones. Nonetheless, it is possible to

apply other compression models, such as the one proposed in [6]. The compression model is orthogonal to the contributions of this work.

4 EXPERIMENTAL EVALUATION

In this section we evaluate the proposed indexing, quantization, and distributed k NN querying algorithms in terms of classification accuracy and query performance. When evaluating the query speed of the k NN query with QED quantization, we set $p = \hat{p}$ as described in Equation 13. In our evaluations we use two distance metrics with QED: QED with Manhattan distance (QED-M), and QED with Hamming distance (QED-H).

4.1 Experimental Setup

We implemented the proposed index and query algorithms in Java, and used the Java API provided by Apache Spark to run our algorithms on an in-house Spark/Hadoop cluster. The Java version installed on the cluster nodes was 1.7.0_79, Spark version 1.6.1. and Hadoop version 2.4.0.

Our Hadoop stack installation is built on the following hardware: There is one Namenode (master) server (Two 6-core Intel Xeon E5-2420v2, 2.2GHz, 15MB Cache; 48 GB RAM at 1333 MT/s Max). The cluster also contains four Datanode (slave) servers (two 6-core Intel Xeon E5-2620v2, 2.1 GHz, 15MB Cache; 64 GB RAM at 1333 MT/s Max). As cluster resource manager we used Apache Yarn. The namenode and datanodes are connected to each other over 1 Gbps Ethernet links over a dedicated switch. Unless otherwise noted, we use all the available hardware resources in this cluster for running the experiments.

In our experiments we used a number of real datasets to evaluate the proposed indexing and querying. We used nine datasets from the UCI repository [4] for accuracy evaluation, and two larger datasets (HIGGS[2], and Skin Data [32]) were used on the Spark/Hadoop cluster for performance measurements. The number of dimensions in the datasets range from 19 to 279 and the number of classes from 2 to 24. The Skin-Images dataset contains integer numbers (image pixel values), while the other datasets contain real numbers. The details of the characteristics of the data and the class distribution can be found in Table 1.

Dataset	Rows	Cols	Classes
anneal	798	38	5
arrhythmia	452	279	13
dermatology	366	33	6
higgs	11M	28	2
horse-colic	300	26	2
ionosphere	351	33	2
musk	476	165	2
segmentation	210	19	7
skin-images	35M	243	2
soybean-large	307	34	19
wdbc	569	30	2

Table 1: Description of the characteristics of the real datasets used in the experiments.

4.2 QED Classification Accuracy

Classification accuracy is computed over the labeled data using the leave-one-out methodology as the number of correct classifications divided by the total number of tuples in the data set. Voting was used to decide the class for each data point. Table 2 shows the best classification accuracy when using k NN classification

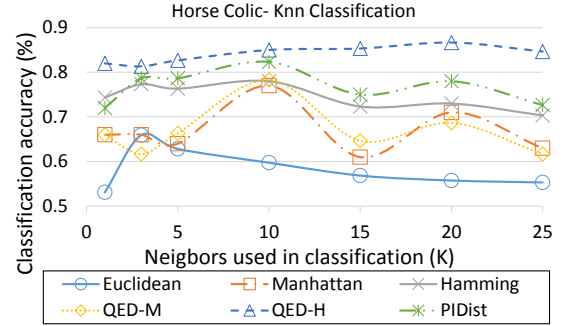


Figure 7: k NN Classification accuracy as the number of nearest neighbors (k) increases for Horse Colic dataset. (Dataset: HorseColic, 300 rows, 26 attributes, 2 classes (99/201))

for each method. We vary the number of nearest neighbors used in classification $k = \{1, 3, 5, 10\}$, and report the best result for each distance function. For quantization, we apply Equi-width and Equi-Populated partitioning varying the number of bins/clusters from 3 to 20 $\{3, 5, 7, 10, 15, 20\}$. The same number of bins/clusters was used for all the dimensions. The only case where attributes could be quantized using a different number of bins/clusters than the one provided as a parameter was the categorical attributes with less categories than the number of bins/clusters provided. In that case each value was considered as a bin/cluster. For dynamic quantization we set p as a percentage of the number of rows. We vary $p = \{60\%, 50\%, 40\%, 30\%, 25\%, 20\%, 10\%, 5\%, \text{ and } 1\%\}$.

For each function metric we report the best result, and then the best accuracy for each dataset is highlighted in bold in Table 2. As shown in the table, QED is able to improve the results for Manhattan and Hamming in most datasets. QED using Manhattan is consistently better than Manhattan with no quantization (8/9) with up to 7.35% accuracy increase (2.4% on average). For Hamming distance, QED quantization outperformed no-quantization in 7/9 cases with up to 57.7% accuracy improvements (10.95% on average).

4.2.1 Evaluation of Parameter k . When using nearest neighbor searches for classification purposes, the number of neighbors considered is often crucial for the accuracy of the classifier. In this experiment we evaluate the effect of k (the number of nearest neighbors) when QED is used in k -Nearest Neighbor (k NN) classification.

Figures 7 and 8 show the classification accuracy for several distance functions as the number of neighbors k increases for two different datasets. In figure 7 for the Horse-colic dataset, the classification accuracy increases gradually for QED (QED-M with Manhattan distance, and QED-H with Hamming distance), while the other distance functions are more sensitive to the value of k . Regardless of the value picked for k , QED-H has the highest accuracy among the measured distance functions for k NN classification for this dataset.

As Figure 8 shows for the Arrhythmia dataset, QED-M (with Manhattan distance) has the highest accuracy. It is worth noting that while the accuracy performance for other distance functions decreases as k increases, classification accuracy for QED is not significantly affected.

4.2.2 Evaluation of Parameter p . The p parameter determines the number of tuples for each dimension to be considered similar for distance computation, while the other tuples get a dissimilarity penalty. Expressed as a percentage, p falls within the

Dataset	Euclidean	Manhattan	QED-M	Hamming			QED-H	PiDist/iGrid	
				NQ	EW	ED		EW	ED
anneal	.934	.939	.964	.986	.984	.980	.994	.990	.990
arrhythmia	.659	.653	.701	.602	.686	.646	.650	.695	.635
dermatology	.975	.978	.986	.975	.973	.883	.921	.981	.970
horse-colic	.740	.770	.783	.780	.827	.857	.867	.833	.843
ionosphere	.866	.909	.943	.809	.926	.860	.920	.929	.903
musk	.882	.893	.916	.819	.876	.870	.878	.868	.887
segmentation	.843	.886	.881	.586	.871	.857	.924	.900	.876
soybean	.873	.899	.938	.909	.912	.902	.821	.909	.922
wdbc	.940	.949	.949	.692	.967	.951	.967	.961	.960

Table 2: Leave-one-out best classification accuracy using k -nearest neighbor ($k \in \{1, 3, 5, 10\}$) classification with different distance functions and quantization methods (NQ=No Quantization, EW=Equi-width, ED=Equi-depth, QED=Query-dependent Equi-depth). The best result for each dataset is highlighted in bold.

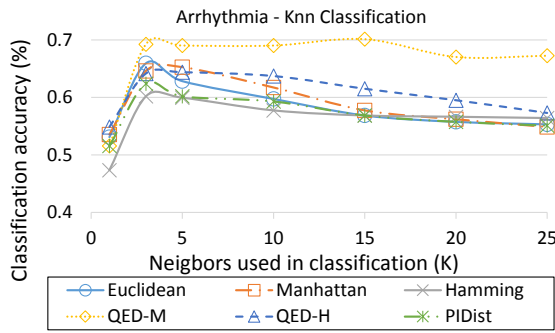


Figure 8: k NN Classification accuracy as the number of nearest neighbors (k) increases for Arrhythmia dataset. (Dataset: Arrhythmia, 452 rows, 279 attributes, 13 classes)

interval $(0, 1]$. If $p = 1$ then the results of the k NN query are the same for both QED-M and Manhattan distance. Clearly, the value of p affects the accuracy of the k NN query results.

In this experiment we vary the value of p from 0.01 to 0.6 and measure the k NN classification accuracy. We chose the two largest of the datasets: HIGGS and Skin-Images, as a higher number of data objects results in a more robust evaluation of p . The accuracy is reported after running 1000 queries obtained by random sampling. We compare the k NN classification accuracy results of QED against sequential scan Manhattan distance and a distributed implementation of Locality Sensitive Hashing (LSH).

The LSH implementation and parameter choice was largely based on the description in chapter 3 of [28]. The LSH number of bins was set to 10000, number of hash functions: 25, and the number of hash tables: 4. For all three methods 5 nearest neighbors were considered for classification.

Figures 9 and 10 show the k NN classification accuracy results as the value of parameter p varies. The filled marker is the p value computed using Equation 13 from Section 3.5.1, and in both cases it is at, or near the highest accuracy point.

4.3 Index size

Given the rapid advancements in data collection, not only the data becomes more complex and harder to analyze, but also its size requires more computational resources. As described earlier, we make use of the BSI index to represent data in a more compact form. A smaller index size should enable performance gains through less network shuffling, fewer CPU cycles required for processing, and less memory utilization and I/O.

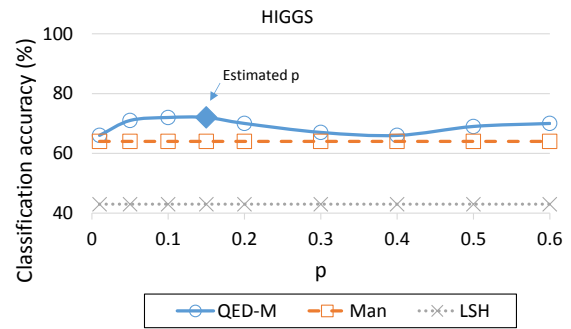


Figure 9: The impact of the p parameter on k NN classification accuracy (Dataset: HIGGS, 11M rows, 28 attributes, 2 classes)

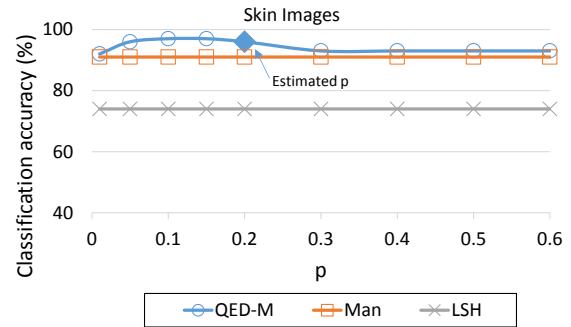


Figure 10: The impact of the p parameter on k NN classification accuracy (Dataset: Skin-Images, 35M rows, 243 attributes, 2 classes)

An important advantage of the BSI index is that it has a compact size. The compression comes not only from using a lower number of bit-slices per attribute than the number of bits used in a Long or Double data type, but also from compressing each individual bit-slice (where beneficial) using a hybrid bitmap compression scheme [14]. The compression of the bit-slices occurs only if it can improve the query performance.

Figure 11 shows the size of the BSI index in comparison with the size of the raw data, the LSH index, and the PiDist (10 and 20) index for the HIGGS and Skin Images datasets. Five LSH hash tables were generated using 25 hash functions and 10,000 bins. PiDist-10 refers to the PiDist index with the bin size of 10, while

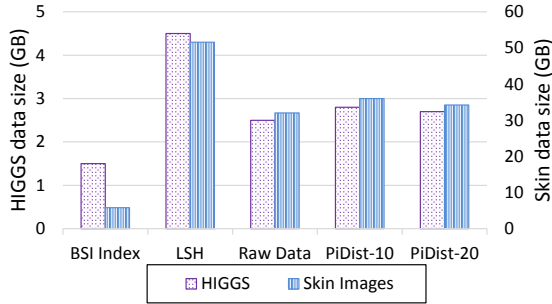


Figure 11: Index sizes for the HIGGS and Skin-Images datasets.

PiDist-20 has the bin size of 20. These are some of the bin sizes that where shown to perform well in [1].

The Skin Images dataset has a higher compression ratio than HIGGS when compared to the raw data size. This is mostly due to the low cardinality of this dataset, which has RGB encoded pixel values. The HIGGS dataset requires approximately 60 slices per attribute to encode its values, while the Skin Images dataset only requires 8 bit-slices per attribute (values from 0 to 255).

Because the BSI index does not require accesses to the raw data, and due to its small size, it is possible to fit more information into memory and less network communication is required when the k NN queries are performed in a distributed setting.

4.4 Performance impact of data cardinality

Given that the BSI index is sensitive to data cardinality, we set up to measure the scalability of the QED quantization method when compared to running NN searches over the BSI index without QED quantization. We use the HIGGS dataset for this experiment as its data has high cardinality. We vary the number of bit-slices per attribute for indexing from 15 to 60. Note that while it is possible to encode any attribute with any number of slices, using less than $\lceil \log_2 c_i \rceil$ slices, where c_i is the attribute cardinality, results in a lossy compression where the values are approximated to some degree. This approximation however, could have little effect on the k NN classification accuracy depending on the dataset. The evaluation of the BSI approximation is left as a subject for future work.

The query time is reported in milliseconds per query, and was obtained by averaging the k NN classification query times over 1000 queries. As Figure 12 shows, with the increase in cardinality, the query speed degrades at a much slower pace for QED-M than BSI Manhattan (without QED quantization). As mentioned in the previous section when describing Algorithm 2, the performance improvements is largely due to a smaller output, independent of the attribute cardinality, and consequently less data shuffling and processing in the aggregation phase. Running the same queries with Manhattan distance without any indexing took approximately two seconds. Thus, the k NN query time using BSIs was two to five times faster than sequential scan, while QED-M achieved an improvement in performance of one order of magnitude.

4.5 QED query time performance

Many of the existing indexing techniques fail to run faster than sequential scan when tested against high dimensional data. Thus, the approximate nearest neighbor searches became a solution that improves on query time performance by trading off accuracy.

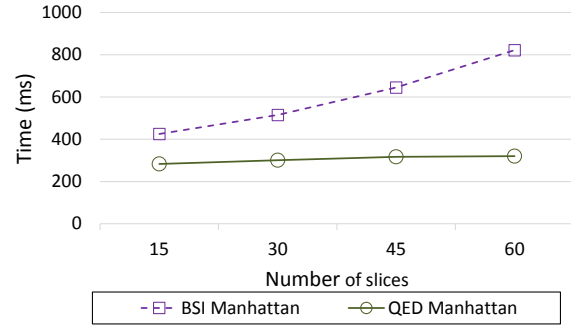


Figure 12: BSI Manhattan and QED Manhattan k NN query performance when increasing data cardinality (Datasets: HIGGS)

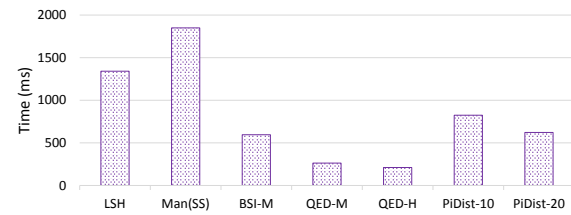


Figure 13: k NN query performance comparison. (Dataset: HIGGS, 11M rows, 28 attributes, 2 classes)

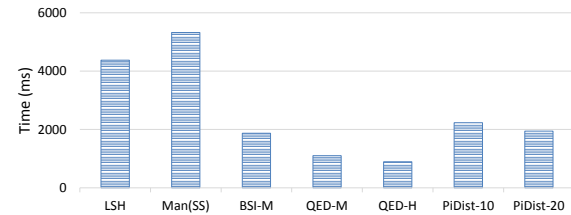


Figure 14: k NN query performance comparison. (Dataset: Skin-Images, 35M rows, 243 attributes, 2 classes)

We ran a total of 1000 k NN classification queries over the HIGGS and Skin Images datasets, with the configuration for LSH and PiDist described earlier. Figures 13 and 14 show the average query time per query in milliseconds. Because the number of nearest neighbors k in k NN classification applications is generally low, we set k in our query speed evaluations to 5 and do not vary it. Increasing k , however, doesn't impact the query performance in any significant way because the scores are computed for all the points in the dataset regardless of k . The best query times were achieved when using the QED quantization over the BSI index. The average query time for QED-M was only 14% of the average Sequential Scan query time for the HIGGS dataset. For the Skin Images data set the QED Manhattan query time was 20% of Sequential Scan query time.

5 CONCLUSION

In this work we described the indexing structure and the methods for on-the-fly Query dependent Equi-Depth (QED) quantization to improve high-dimensional similarity. The quantization is done for each dimension at query time and localized scores are generated for the closest $p\%$ of the points. A constant penalty is applied for the rest of the points. By normalizing the penalty for values outside the similarity range we are able to improve nearest neighbor searches in high dimensional spaces.

We evaluated the kNN classification accuracy of the proposed QED quantization on a set of nine high-dimensional datasets and observed an average improvement in accuracy of 2.4% for Manhattan distance and 10.95% for Hamming distance when using QED quantization.

The index structure and the query algorithms that support the kNN searches were designed with distributed processing in mind. We implemented several BSI arithmetic operations such as: addition with a constant, absolute value, and various transformation of the BSI attribute. The index can be partitioned vertically as well as horizontally and makes for a fine level of task granularity and load balancing. Because each dimension is indexed independently, this approach is also scalable for high dimensional data. We evaluated the scalability for datasets up to 243 dimensions on a Spark/Hadoop cluster. We also show that when using QED quantization, the kNN query performance is very robust and does not decrease significantly with the increase in data cardinality.

Due to a smaller index size, the ability to partition the index vertically and horizontally, and the fast bitwise operations, the BSI index proves to be a good data structure for performing distributed Nearest Neighbor searches with query aware quantization at run time. With QED quantization, in our performance evaluation we observe an improvement in query time of up to one order of magnitude when compared to Sequential Scan on high dimensional data.

As future work we plan to investigate further the penalty applied for dissimilar dimensions and under what conditions the normalization of the penalty or the distance would improve the accuracy of nearest neighbor searches. More work is also required in expanding the distance metrics for which the QED quantization can be applied.

6 ACKNOWLEDGEMENTS

This work was partially supported by the NIH National Cancer Institute/Big Data to Knowledge (BD2K) Program under grant R01CA214825 and joint NSF/NIH Initiative on Quantitative Approaches to Biomedical Big Data (QuBDD) (R01) grants NSF DMS-1557578 and NIH R01CA225190.

REFERENCES

- [1] C. C. Aggarwal and P. S. Yu. The igrind index: reversing the dimensionality curse for similarity indexing in high dimensional space. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 119–129. ACM, 2000.
- [2] P. Baldi, P. Sadowski, and D. Whiteson. Searching for exotic particles in high-energy physics with deep learning. *Nature Commun*, 5, 2014.
- [3] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is “nearest neighbor” meaningful? In *International conference on database theory*, pages 217–235. Springer, 1999.
- [4] C. Blake and C. Merz. Uci repository of machine learning databases [http://www.ics.uci.edu/mllearn/mlrepository.html], department of information and computer science. *University of California, Irvine, CA*, 1998.
- [5] O. Boiman, E. Shechtman, and M. Irani. In defense of nearest-neighbor based image classification. In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pages 1–8. IEEE, 2008.
- [6] S. Chambi, D. Lemire, O. Kaser, and R. Godin. Better bitmap performance with roaring bitmaps. *arXiv preprint arXiv:1402.6407*, 2014.
- [7] D. L. Donoho et al. High-dimensional data analysis: The curses and blessings of dimensionality. *AMS Math Challenges Lecture*, 1:32, 2000.
- [8] M. Ester, H.-P. Kriegel, J. Sander, X. Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231, 1996.
- [9] U. Fayyad and K. Irani. Multi-interval discretization of continuous-valued attributes for classification learning. In *Joint Conference on Artificial Intelligence*, pages 1022–1027, 1993.
- [10] J. Gao, H. V. Jagadish, W. Lu, and B. C. Ooi. Dsh: data sensitive hashing for high-dimensional k-nnsearch. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1127–1138. ACM, 2014.
- [11] S. García, J. Luengo, J. A. Sáez, V. López, and F. Herrera. A survey of discretization techniques: Taxonomy and empirical analysis in supervised learning. *IEEE Trans. Knowl. Data Eng.*, 25(4):734–750, 2013.
- [12] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases*, pages 518–529. Morgan Kaufmann Publishers Inc., 1999.
- [13] A. Guttman. *R-trees: a dynamic index structure for spatial searching*, volume 14. ACM, 1984.
- [14] G. Guzun and G. Canahuate. Hybrid query optimization for hard-to-compress bit-vectors. *The VLDB Journal*, pages 1–16, 2015.
- [15] G. Guzun and G. Canahuate. Supporting dynamic quantization for high-dimensional data analytics. In *Proceedings of the ExploreDB’17, ExploreDB’17*, pages 6:1–6:6. New York, NY, USA, 2017. ACM.
- [16] G. Guzun, G. Canahuate, and D. Chiu. A two-phase mapreduce algorithm for scalable preference queries over high-dimensional data. In *Proceedings of the 20th International Database Engineering & Applications Symposium, IDEAS*, 2016.
- [17] G. Guzun, G. Canahuate, D. Chiu, and J. Sawin. A tunable compression framework for bitmap indices. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 484–495. IEEE, 2014.
- [18] G. Guzun, J. C. McClurg, G. Canahuate, and R. Mudumbai. Power efficient big data analytics algorithms through low-level operations. In *Big Data (Big Data), 2016 IEEE International Conference on*, pages 355–361. IEEE, 2016.
- [19] G. Guzun, J. Tosado, and G. Canahuate. Slicing the dimensionality: Top-k query processing for high-dimensional spaces. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems XIV*, pages 26–50. Springer, 2014.
- [20] Q. Huang, J. Feng, Y. Zhang, Q. Fang, and W. Ng. Query-aware locality-sensitive hashing for approximate nearest neighbor search. *Proceedings of the VLDB Endowment*, 9(1):1–12, 2015.
- [21] H. Jegou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128, 2011.
- [22] J. Johnson, M. Douze, and H. Jégou. Billion-scale similarity search with gpus. *arXiv preprint arXiv:1702.08734*, 2017.
- [23] I. Kamel and C. Faloutsos. Hilbert r-tree: An improved r-tree using fractals. Technical report, 1993.
- [24] N. Katayama and S. Satoh. The sr-tree: An index structure for high-dimensional nearest neighbor queries. In *ACM SIGMOD Record*, volume 26, pages 369–380. ACM, 1997.
- [25] R. Kerber. Chimerge: Discretization of numeric attributes. In *Proceedings of the 10th National Conference on Artificial Intelligence. San Jose, CA, July 12-16, 1992.*, pages 123–128, 1992.
- [26] L. A. Kurgan and K. J. Cios. Caim discretization algorithm. *IEEE Transactions on Knowledge and Data Engineering*, 16(2):145–153, 2004.
- [27] D. Lemire, O. Kaser, and E. Gutarra. Reordering rows for better compression: Beyond the lexicographic order. *ACM Transactions on Database Systems*, 37(3):20:1–20:29, 2012.
- [28] J. Leskovec, A. Rajaraman, and J. D. Ullman. *Mining of massive datasets*. Cambridge university press, 2014.
- [29] F. Liu and H. J. Lee. Use of social network information to enhance collaborative filtering performance. *Expert systems with applications*, 37(7):4772–4778, 2010.
- [30] P. O’Neil and D. Quass. Improved query performance with variant indexes. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 38–49. ACM Press, 1997.
- [31] V. Pareto. Manual of political economy, 1906.
- [32] S. L. Phung, A. Bouzerdoum, and D. Chai. Skin segmentation using color pixel classification: analysis and comparison. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(1):148–154, Jan 2005.
- [33] D. Rinfret. Answering preference queries with bit-sliced index arithmetic. In *Proceedings of the 2008 C 3 S 2 E conference*, pages 173–185. ACM, 2008.
- [34] D. Rinfret, P. O’Neil, and E. O’Neil. Bit-sliced index arithmetic. In *ACM SIGMOD Record*, volume 30, pages 47–57. ACM, 2001.
- [35] D. Rinfret, P. O’Neil, and E. O’Neil. Bit-sliced index arithmetic. *SIGMOD Rec.*, 30(2):47–57, 2001.
- [36] H. Samet. *The design and analysis of spatial data structures*, volume 199. Addison-Wesley Reading, MA, 1990.
- [37] K. shy Goh, B. Li, and E. Chang. Dyndex: A dynamic and non-metric space index. In *IN ACM MULTIMEDIA*, pages 466–475, 2002.
- [38] C. Tsai, C. Lee, and W. Yang. A discretization algorithm based on class-attribute contingency coefficient. *Inf. Sci.*, 178(3):714–731, 2008.
- [39] A. K. H. Tung, R. Zhang, N. Koudas, and B. C. Ooi. Similarity search: a matching based approach. In *VLDB’2006: Proceedings of the 32nd international conference on Very large data bases*, pages 631–642. VLDB Endowment, 2006.
- [40] K. Wu, E. J. Otoo, and A. Shoshani. Compressing bitmap indexes for faster search operations. In *Proceedings of the 2002 International Conference on Scientific and Statistical Database Management Conference (SSDBM’02)*, pages 99–108, 2002.
- [41] K. Wu, E. J. Otoo, A. Shoshani, and H. Nordberg. Notes on design and implementation of compressed bit vectors. Technical Report LBNL/PUB-3161, Lawrence Berkeley National Laboratory, 2001.

Global-Scale Placement of Transactional Data Stores

Victor Zakhary¹, Faisal Nawab², Divyakant Agrawal¹, Amr El Abbadi¹

¹University of California, Santa Barbara, CA 93106

[victorzakhary, agrawal, amr]@cs.ucsb.edu

²University of California, Santa Cruz, CA 95064

fnawab@ucsc.edu

ABSTRACT

Global-Scale Data Management (GSDM) empowers systems by providing higher levels of fault-tolerance, read availability, and efficiency in utilizing cloud resources. But, *at which datacenters should data be placed?* Current cloud providers offer tens of datacenters and hundreds of edge datacenters that are globally distributed all over the world. Unlike networks within a datacenter, the topology of the Wide-Area Network (WAN) is asymmetric and diverse—the latency connecting a pair of datacenters can be an order of magnitude larger than the latency connecting another pair. This makes placement a significant factor in performance. However, it is not only placement. The specifics of the transaction management protocol play a crucial role in deciding which placement is ideal. In this paper, we develop GPlacer, a placement optimization framework that embeds the transaction protocol constraints into an optimization to derive both the data placement and the transaction protocol configuration that minimize the overall transaction latency. In developing GPlacer, we discover counter-intuitive lessons about data placement and transaction execution practices. Our evaluation shows that applying these lessons in addition to known best practices generate deployments that reduce the average transaction latency by up to 68%.

1 INTRODUCTION

Internet applications strive for high-performance 24/7 service to clients dispersed around the world. Achieving this is threatened by complete datacenter outages; either planned or unplanned. To overcome these challenges, application services and their backend databases are increasingly being deployed on multiple datacenters spanning large geographic regions (*geo-replication*). F1 [29], Spanner [11], and Tao [8] are examples of deployed systems that are geographically replicated for fault-tolerance and performance reasons.

Moving to Global-Scale Data Management (GSDM), despite its benefits, raises many challenges that are not faced by traditional deployments. The large WAN communication latency is orders of magnitude larger than the traditional LAN communication latency. Figure 1 illustrates the latency difference between communication messages that occur within the same machine, among different machine in the same datacenter, or in multiple datacenters in different geographical regions. This large communication latency of the WAN motivates systems like Yahoo’s PNUTS [9], Facebook’s Tao [8] and others [17, 24] to trade off replica consistency and/or multi-row transaction support with high availability and scalability. However, enterprise applications and applications with complex and evolving schemas have more interest in data management systems that provide transactional

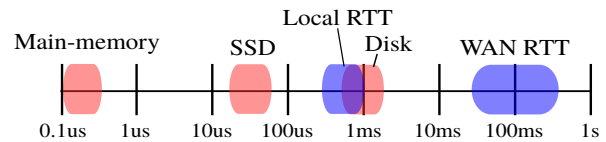


Figure 1: Latency of the Wide-Area Network Round-Trip Time communication (WAN RTT) compared to memory access latency [27] and network latency within the datacenter (local RTT).

ACID properties [5, 11, 30]. Application developers spend significant time to build transaction semantics and complex mechanisms, that are error-prone, on top of the eventual consistent datastores in order to handle stale data items and reason about inconsistency [11, 29]. Therefore, in the past few years, many solutions have emerged to provide strongly consistent transactions for geo-replicated databases [11, 16, 18, 21–23]. These solutions use different replication and isolation techniques in order to minimize the number of WAN messages required to achieve strong ACID transactional guarantees for geo-replicated databases, hence reducing the transaction latency.

Data placement is the problem of deciding the subset of datacenters to host a full or a partial replica of the data to achieve a certain objective such as minimizing the transaction latency, minimizing the deployment monetary costs, and any combination of these and other user-defined objective functions.

In this paper, we propose GPlacer; an optimization framework that solves the data placement problem. GPlacer embeds the commit protocol constraints into an optimization to derive both the data placement and the commit protocol configurations that minimize the overall transaction latency. In developing GPlacer, we discover counter-intuitive lessons about data placement and transaction execution practices. These lessons exploit the latency diversity and asymmetry of the WAN links and are widely applicable to Paxos-based commitment protocols [13, 21] and leader-based commitment protocols [5, 11]. GPlacer incorporates these lessons, the commitment protocol constraints, and the application requirements in an optimization to find the placement that minimizes the average transaction latency.

WAN links are diverse and asymmetric; a link connecting a pair of datacenters can be an order of magnitude larger than a link connecting another pair. Table 1 shows the average measured Round-Trip Time (*RTT*) between every pair of nine Amazon AWS datacenters in California (*C*), Oregon (*O*), Virginia (*V*), São Paulo (*SP*), Ireland (*I*), Sydney (*Sy*), Singapore (*Si*), Tokyo (*T*), and Seoul (*Se*). As shown, the average *RTT* between California and Oregon datacenters is 22ms while the average *RTT* between Singapore and São Paulo datacenters is 329ms. Therefore, the number of WAN messages required per transaction is not the only factor that dominates the transaction latency. Transaction latency is a product of both the transaction commit protocol, which

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26–29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

	C	O	V	I	Si	T	Se	Sy	SP
C	0(1)	22(2)	65(13)	136(5)	189(12)	113(5)	142(12)	159(2)	185(11)
O	22(2)	1(1)	88(14)	125(2)	166(13)	101(11)	131(13)	178(3)	182(11)
V	65(13)	88(14)	1(16)	73(13)	220(22)	156(16)	179(20)	219(13)	121(16)
I	136(5)	125(2)	73(13)	0(0)	180(18)	211(10)	233(14)	301(5)	185(12)
Si	189(11)	166(12)	220(22)	180(17)	1(9)	68(8)	97(13)	169(8)	329(21)
T	113(5)	101(11)	156(18)	211(10)	68(9)	0(3)	32(9)	104(2)	263(15)
Se	142(9)	131(13)	179(20)	233(13)	97(13)	32(10)	1(9)	133(8)	290(16)
Sy	159(2)	178(3)	219(12)	301(5)	169(10)	104(2)	133(8)	1(0)	338(11)
SP	185(13)	182(12)	121(17)	185(13)	329(23)	263(16)	290(18)	338(14)	1(11)

Table 1: The average RTT latencies between different datacenters in milliseconds and the standard deviation inside parentheses.

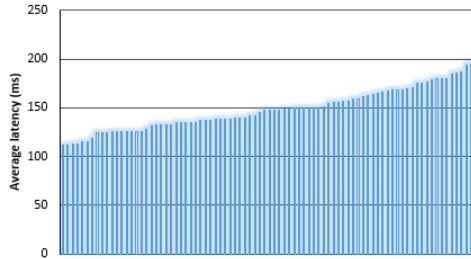


Figure 2: The average latency, of all the clients in 9 datacenters, to reach the closest quorum (2 out of 3) for all the possible $\binom{9}{3} = 84$ different placements sorted by latency.

controls the number of the WAN messages required per transaction, and the locations of the replicas, hence the placement, which controls the latency per a WAN message. To illustrate the placement effect on the average obtained transaction latency, we hold the following experiment. We equally distribute clients among the nine AWS datacenters. Three out of the nine datacenters are chosen to host a data replica. The time to reach the closest quorum, two replicas out of these three, is measured for all the clients for all the **possible placements** and the average latency is reported. Figure 2 shows the effect of only changing the placement on the average obtained latency for all the clients while fixing the protocol. As seen in Figure 2, changing only the placement while fixing all the other parameters (the protocol, the workload distribution, etc.) can lead to a significant change of 1.75x between the minimum and the maximum reported average latency. This latency difference amplifies for real workloads when transactions are executed in chains [20].

Unlike GPlacer that optimizes the placement for *multi-row transactional workload with strong consistency requirements*, many works focus on optimizing the placement for weaker consistency levels and single-row operations. SPANStore [33] develops an optimization framework to optimize the monetary cost of deploying a geo-replicated *key/value* store. This framework optimizes the total cost of processing, storage, bandwidth, and I/O and finds the placement that achieves the minimum overall cost while meeting the application requirements. Liu et al. [19], like SPANStore, optimize the deployment monetary cost. However, they consider cost savings exploiting resource reservation payment model instead of the pay-as-you-go payment model while avoiding over reservation. Ping et al. [26] propose the use of a utility function to derive a placement that achieves a balance between the availability and the speed of data access. Volley [4] analyzes data access logs and

generates a migration plan for data partitions to minimize the access latency.

Sharov et al. [28] optimize the placement for *strong consistent transactions* using *leader-based* protocols. Sharov assumes that a database is sharded into multiple *partitions* and each partition is **replicated** independently. Each partition has a **leader replica** that serializes all the transactions that span this partition to achieve isolation. This leader replicates the updates to a *majority quorum* of the partition replicas to achieve fault tolerance. Although they provide placements for strong consistent transactional workloads, their optimizations are tightly coupled with leader-based protocols and it does not apply to the many non-leader-based protocols that are widely used such as [6, 13, 16, 21, 25]. Also, their resulting optimal placement allocates all the partition leaders together in one datacenter. Placing all partition leaders in one datacenter introduces the risk of losing access to the entire data until the leaders are re-elected. In addition, the transactions that span a single partition might incur higher latency than the latency observed when the leader of each partition is placed closer to the clients that access this partition.

The rest of the paper is organized as follows. Section 2 explains the transaction model, the client requests, and the assumptions and limitations of application requirements. Although GPlacer can optimize the placement for different classes of commitment protocols, a Paxos-based protocol is used to explain the details of GPlacer. Section 3 formalizes the placement problem into an exhaustive search problem. Although the exhaustive search finds the optimal placement, it does not efficiently scale with the number of datacenters. Therefore, we introduce several placement heuristics that find sub-optimal placements while efficiently scale with the number of datacenters. Section 4 describes the counter-intuitive lessons learned during the development of GPlacer and their effect on the transaction latency. In Section 5, we evaluate the effect of the placement lessons on the transaction latency and the abort rate. We also evaluate the output and the performance of the proposed heuristics compared to the exhaustive search. In Section 6, we explain the changes that need to be done to extend GPlacer to optimize for other protocols. The paper is concluded in Section 7.

2 BACKGROUND

Global-scale placement is the problem of deciding which datacenters will store a full or a partial replica of an application’s data subject to a certain objective function. Objective functions can vary between minimizing the deployment monetary cost [19, 33] or minimizing the data access latency for a defined set of client operations [28]. Objective functions are always constrained by the application requirements (e.g., availability, upper bound access time, or bandwidth usage). In this section, we present our storage model and our assumptions about the workload distribution, the application requirements, and the objective function.

The universe of datacenters, denoted by DC , is defined as all the datacenters that can host an application¹ instance and/or a replica² of the database. We assume that the application is deployed on a subset of the datacenters $DC_{app} \subseteq DC$. The clients of the application are scattered around the globe and for simplicity, we assume that clients are collocated with their closest datacenter. The application is deployed in all the datacenters that have clients. However, these datacenters can be different

¹Application refers to the middle tier logic.

²Replica refers to a copy of the backend database.

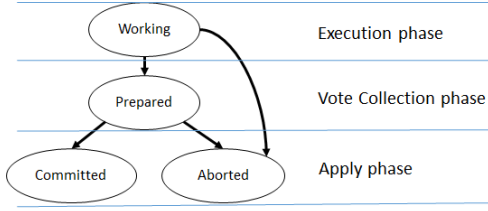


Figure 3: The state-transition diagram of a distributed transaction adopted from [13].

from the datacenters that host replicas. $DC_{db} \subseteq DC$ is the subset of datacenters that host a database replica. We assume that the database is *partitioned* and all the partitions are *fully replicated* in DC_{db} .

2.1 The Transaction Management Protocol

The clients of the application access the globally-distributed storage by issuing transactions, which are collections of read and write operations followed by a commit or an abort. GPlacer considers transactions with *strong guarantees*, i.e., serializability [7]. Strong consistent transactions on globally-distributed data require more coordination than weaker forms of access like eventual consistency or single-key atomicity— thus making strongly consistent transactions more expensive. A strong consistency transactional interface is more natural to programmers and is required by many applications. Thus, we adopt such strong access semantics for GSDM as others did from both academia [16, 21] and industry [11].

We adopt the distributed transaction model proposed by Gray and Lamport [13]. Figure 3 shows the different states of a transaction and the corresponding execution phases. A client drives the execution of a transaction in three phases. The details of these three phases differ across different transaction management protocols. However, the abstract semantics behind these phases are the same for all the protocols that provide the same strong transactional guarantees. The three phases of a transaction are: the execution phase, the vote collection phase, and the apply phase.

During the execution phase, the transaction is in the working state when read and write operations are processed. We assume that writes are locally buffered at the client and the updates are sent to the data replicas in the second phase. This assumption is widely used in many geo-replicated transaction management protocols [11, 16, 21]. For a read operation, clients communicate with their read coordinator, r_c . r_c processes a read request and responds back to the client. The RTT between a client c and r_c is denoted by RTT_{c-to-r_c} and the time for r_c to process the read request is denoted by P_{r_c} . The total execution phase latency is denoted by $L_e = n_r \cdot (RTT_{c-to-r_c} + P_{r_c})$ where n_r is the average number of read requests per transaction. The transaction management protocol determines the values of n_r , RTT_{c-to-r_c} , and P_{r_c} . Some protocols assume that the client is the read coordinator. In such case, $RTT_{c-to-r_c} = 0$. Also, some protocols require that the client issues read requests one by one and others require that the client should batch all the reads in one request. The processing time P_{r_c} depends on how many replicas r_c should communicate with to serve a read request. In our model protocol, r_c has to communicate with a majority quorum to serve each read (we also consider read optimizations later in Section 2.2. As write requests are locally buffered, their effect on the execution phase latency is negligible. During the execution phase, a client might

decide to abort the transaction by simply moving the transaction to the aborted state. However, if the client decides to commit the transaction, the transaction is moved to the prepared state and the vote collection phase starts.

During the vote collection phase, the client sends the transaction’s details to the commit coordinator c_c which is responsible for coordinating with the other replicas to decide either to commit or to abort the transaction. Typically, the c_c uses either two-phase commit (2PC) with two-phase locking (2PL) [11] or quorum-based approaches (e.g., Paxos) with 2PL [21]. The vote collection phase can be mapped to the first phase of the 2PC or the first round of Paxos. The latency of the voting phase is denoted by $L_v = \frac{RTT_{c-to-c_c}}{2} + RTT_{c_c-to-p}$ where RTT_{c-to-c_c} is the RTT between c and c_c and RTT_{c_c-to-p} is the round-trip time between the c_c and the furthest participant p included in the voting process.

If the decision of the vote collection phase is to abort, the client is notified, the transaction is moved to the aborted state, the other participants are asynchronously updated, and the obtained locks are released. However, if the decision is to commit, c_c starts the apply phase by sending the apply message to all the participants. Upon receiving the apply message, the participants commit the transaction, release the locks, and respond back to the coordinator. c_c notifies the client and the transaction is moved to the committed state. The latency of the apply phase is denoted by $L_a = RTT_{c_c-to-p} + \frac{RTT_{c-to-c_c}}{2}$ as the apply phase takes a round of communication with the participants in addition to the time to inform the client about the decision. A transaction commit latency L_c is the time spent in the vote collection phase and the apply phase combined: $L_c = RTT_{c-to-c_c} + 2 \cdot RTT_{c_c-to-p}$. The transaction latency L_t is the time from the beginning till the end of a transaction: $L_t = L_e + L_c = n_r \cdot (RTT_{c-to-r_c} + P_{r_c}) + RTT_{c-to-c_c} + 2RTT_{c_c-to-p}$.

GPlacer optimizes the average overall transaction latency over all the clients in different datacenters. It is designed to optimize the placement for a wide class of transaction commitment protocols. In this paper, we focus on optimizing for multi-master Paxos-based protocols [13, 21] and for leader-based protocols [11] both on *partitioned fully replicated databases*. In multi-master Paxos, each replica can act as the commitment coordinator role and uses the two rounds of Paxos for both transaction isolation and replication. However, in leader-based protocols, a transaction can fall into one of two categories: single-partition transactions or multi-partition transactions. Single-partition transactions span only one partition and the isolation between transactions that span this partition is managed by the leader of this partition. Multi-partition transactions span multiple partitions and typically 2PC is used between the leaders of the partitions involved in a transaction to achieve isolation. In both categories, partition leaders replicate the updates of committed transactions to a majority quorum of their partition replicas using only the second round of Paxos.

In Section 3, we formalize GPlacer. We use Replicated Commit [21] as our protocol model where reads are served from a majority of the replicas and commits are done using the two rounds of Paxos for isolation and replication. In Section 2.2, we explain some commonly used optimization to reduce the execution phase latency. In Section 6, we explain how to extend GPlacer to optimize placement for leader-based protocols like Spanner [11].

2.2 Read optimizations

In this section, we present two widely-used read optimizations that are considered in GPlacer. A read request latency $L_r = (RTT_{c-t_0-r_c} + P_{r_c})$. The first optimization, **optimistic read**, aims to eliminate the read processing time P_{r_c} . The second optimization, **passive replica read** aims to eliminate the time to reach the coordinator $RTT_{c-t_0-r_c}$ and the processing time P_{r_c} by bringing a copy of the data to the client’s datacenter. We define two different types of replicas a datacenter can host: *active replica* or *passive replica*. An active replica contributes synchronously in the voting collection and the apply phases and can act the coordinator and the participant roles. However, a passive replica is a read-only replica. It is asynchronously updated after the transactions are committed.

Optimistic read aims to eliminate the read request processing time by optimistically reading data values from the closest active replica without any coordination with other active replicas. This optimization has been introduced before as early as in Postgres-R local reads [15] and as fast reads in Zookeeper [14]. Applying optimistic reads require validating the value read in the commit phase to guarantee the freshness of the optimistically read values in the execution phase. In Spanner [11], reads are served by the leader of each partition. However, optimistic reads can be beneficial by reading from the closest partition replica instead from the partition leader. In Replicated Commit [21], a client is required to read from a quorum of the replicas. Applying optimistic read reduces the read latency by reading from one replica instead of a quorum.

Passive replica read aims to completely eliminate the read latency by processing read requests from a local read-only replica or a **passive replica**. The reason behind this naming is that a passive replica does not participate actively in the commit decision. Therefore, adding more passive replicas does not affect the commit latency. However, these replicas need to be asynchronously updated which increases the bandwidth required per committed transaction. Also, having many passive replicas increases the deployment cost. Data read from a passive replica needs to be validated in the commit phase to guarantee freshness. If the data is frequently updated at the active replicas, the data values read from a passive replica will be stale which increases the transaction abort rate. The concept of passive replica read has also been introduced in [28] as *weak reads*.

GPlacer chooses the set of active replicas and the set of passive replicas. In addition, it assigns r_c and c_c for clients in every datacenter. Application requirements are given as inputs to the framework. GPlacer takes as an input the fault tolerance level f , the total number of replicas t , and the workload distribution. f determines the number of active replicas and t determines the number of passive replicas. The workload distribution determines which datacenters should have active replicas, which should have passive replicas, and which should not have a replica at all. GPlacer finds placements that optimize the overall average transaction latency for strongly consistent multi-row transaction workloads. However, systems that require non-transactional or weakly consistent operations can easily be tuned in GPlacer’s prototype but we do not discuss them since they were treated in previous works [4, 33].

3 FRAMEWORK FORMULATION

GPlacer finds the placement that minimizes the average transaction latency for partitioned fully replicated databases. As explained in Section 2, Paxos-based protocols use majority quorums for both transaction isolation and replication while leader-based protocols use majority quorums only for replication. Placement for Paxos-based protocols requires finding the subset of datacenters that should host replicas and the majority quorums used by the protocol. Leader-based protocols requires an additional step of placing the leaders of different database partitions on the replicas chosen in the first step. In Section 3.1, we formulate the placement problem into an exhaustive search model for Paxos-based protocols. This model evaluates all the possible placement combinations and returns one placement that achieves the minimum average transaction latency for a given workload. The model finds the placement $DC_{db} \subseteq DC$ and the majority quorums for each replica in this placement that optimizes the objective function. Although the model finds the optimal placement, due to the model complexity, it does not scale with the number of datacenters *when multiple cloud providers and edge datacenters are considered*. Therefore, in Section 3.2, we introduce two replica-placement heuristics to find placements that are close to optimal among hundreds of datacenters. The performance and the resulting placements of these heuristics are evaluated in Section 5.

3.1 Model formulation

The **inputs** of GPlacer fall into *two* categories:

- **Datacenter information:** this includes the number of datacenters $|DC|$ and the average RTT between every pair of the datacenters.
- **Application information:** this includes the number of datacenter scale outages f the deployment should tolerate and the application workload distribution. The workload distribution is denoted by c_i and represents the number of clients c at datacenter i .

The **outputs** of GPlacer include:

- The list of datacenters that should host a database replica.
- The read and the commit coordinator of clients at each datacenter. Clients at one datacenter share the same read and commit coordinators.

As the placement problem can be represented as an optimization model, we first implemented the placement model as an integer program and used the open source GLPK solver [2]. However, the solver could not efficiently scale with the number of datacenters. Many of the optimization constraints are conditional and to convert them to linear constraints, multiple binary output variables are introduced. The binary outputs and their related constraints are quadratic in the number of the datacenters $O(|DC|^2)$. In addition, GLPK solver introduced performance overhead. Therefore, to conduct a fair comparison with the replica-placement heuristics, we implement both the exhaustive search and the heuristics in Java. The **objective function** of the placement model is to minimize the average transaction latency of all the clients in all the datacenters. Algorithm 1 shows the details of the exhaustive search model.

Algorithm 1 evaluates all the possible subsets of the input datacenters of size $2f + 1$ and returns the one that minimizes the average transaction latency. The function *evalLat*, in line 3,

Algorithm 1 Evaluates all the possible placement combinations and returns the one that achieves the minimum average latency for given application requirements.

Input: $f, |DC|, RTT_{ij} \forall i, j \in DC$ and the Set $C = \{c_i \forall i \in DC\}$
Output: $DC_{db}, DC_{rc},$ and DC_{cc}

```

1:  $DC_{db}, DC_{rc}, DC_{cc} \leftarrow \{\}, minL \leftarrow MaxInt$ 
2: for each Set  $S \subset DC, |S| = 2f + 1$  do
3:    $l, S_{rc}, S_{cc} \leftarrow evalLat(S, RTT, C)$ 
4:   if  $l < minL$  then
5:      $minL \leftarrow l, DC_{db} \leftarrow S$ 
6:      $DC_{rc} \leftarrow S_{rc}, DC_{cc} \leftarrow S_{cc}$ 
7:   end if
8: end for

```

has different implementations based on the enabled read optimizations. When all the read optimizations are disabled, *evalLat* assumes that the read coordinator and the commit coordinator are collocated with the client who issues a transaction and reads are served from a majority quorum of replicas. However, if *optimistic read* is enabled, the read latency is updated to the RTT to the closest chosen replica from the client. Also, if *passive replica read* is enabled, the read latency is updated to zero as all the clients perform read operations from a local replica.

3.2 Replica-placement heuristics

Although Algorithm 1 finds the optimal placement among all the possible placements, it does not efficiently scale when the total number of the datacenters, $|DC|$, or the number of the replicas, $|DC_{db}|$, increases. Our experiments show that choosing 7 replicas out of 60 datacenters ($\binom{60}{7}$) takes 2 hours while choosing 7 replicas out of hundreds of datacenters (which is the case when we consider edge datacenters) could take years. Therefore, we present two replica-placement heuristics that efficiently find placements with sub-optimal average transaction latency. These replica-placement heuristics consider the *two main aspects* that affect the transaction latency; the latency between the clients and the replicas and the latency between the replicas each other. The running-time of these heuristics is polynomial in the total number of the datacenters. The performance and the resulting placements of these heuristics are compared to the exhaustive search results in Section 5.2.

The first replica-placement heuristic is shown in Algorithm 2. It uses an iterative greedy algorithm to choose the replicas. It starts with an empty set of chosen replicas $DC_{db} \leftarrow \{\}$, line 1, and at each iteration, it adds one replica to DC_{db} until $2f + 1$ replicas are chosen. The inner loop, lines 4-14, evaluates the effect of adding each unchosen replica to DC_{db} on the average transaction latency and the replica that achieves the minimum latency is added to DC_{db} , line 15. *evalLat* is the same evaluation function introduced in Algorithm 1 line 3. The intuition behind this heuristic is that choosing the best candidate at each step should lead to a solution that is optimal or close to the optimal.

The second replica-placement heuristic is presented in Algorithm 3. It is based on the K-Means algorithm. It assigns weights to every datacenter, initially equals to the number of clients in this datacenter; line 4. A datacenter weight is updated according to the number of quorums it participates at; line 13. Datacenter weights are iteratively updated and datacenters are sorted by their weights. The top $2f + 1$ datacenters are chosen to host replicas in lines 5 and 18. The algorithm evaluates the placement

Algorithm 2 Greedily adds one replica at a time achieving the minimum average latency at each iteration.

Input: $f, |DC|, RTT_{ij} \forall i, j \in DC$ and the Set $C = \{c_i \forall i \in DC\}$
Output: $DC_{db}, DC_{rc},$ and DC_{cc}

```

1:  $DC_{db}, DC_{rc}, DC_{cc} \leftarrow \{\}$ 
2: while  $|DC_{db}| < 2f + 1$  do
3:    $S \leftarrow DC_{db}, minL \leftarrow MaxInt, minDC \leftarrow \phi$ 
4:   for all  $dc \in DC$  do
5:     if  $dc \notin S$  then
6:        $S \leftarrow S \cup \{dc\}$ 
7:        $l, S_{rc}, S_{cc} \leftarrow evalLat(S, RTT, C)$ 
8:       if  $l < minL$  then
9:          $minL \leftarrow l, minDC \leftarrow dc$ 
10:         $DC_{rc} \leftarrow S_{rc}, DC_{cc} \leftarrow S_{cc}$ 
11:       end if
12:      $S \leftarrow S \setminus \{dc\}$ 
13:   end if
14:   end for
15:    $DC_{db} \leftarrow DC_{db} \cup \{minDC\}$ 
16: end while

```

in every iteration and stops when the average transaction latency converges. To avoid fast convergence to a local minimum, a minimum iteration count is required before terminating the algorithm; lines 1 and 7. The minimum evaluated placement is saved to make sure that the final placement does not achieve higher transaction latency than any placement that has been evaluated before.

Algorithm 3 Assigns weights to datacenters and iteratively chooses the top weighted $2f + 1$ to host replicas.

Input: $f, |DC|, RTT_{ij} \forall i, j \in DC, t$ and the Set $C = \{c_i \forall i \in DC\}$
Output: $DC_{db}, DC_{rc},$ and DC_{cc}

```

1:  $minIter \leftarrow t, iter \leftarrow 0$ 
2:  $DC_{db}, DC_{rc}, DC_{cc} \leftarrow \{\}$ 
3:  $l_{n-1}, l_n \leftarrow MaxInt$ 
4:  $Weights \leftarrow \{c_0, c_1, \dots, c_{|DC|}\}$  // Initialize weights with the
   number of clients at each datacenter.
5:  $DC_{db} \leftarrow top(sort(Weights), 2f + 1)$  // Sort on weights and
   choose a placement of the top  $2f + 1$ .
6:  $l_n, DC_{rc}, DC_{cc} \leftarrow evalLat(DC_{db}, RTT, C)$ 
7: while  $l_n < l_{n-1} || iter ++ < minIter$  do
8:    $l_{n-1} \leftarrow l_n$ 
9:    $NewW \leftarrow \{0, 0, \dots, 0\}$  // New Weights
10:  for all  $dc1 \in DC$  do
11:    for all  $dc2 \in DC$  do
12:      if  $dc2 \in nearestQuorum(dc1)$  then
13:         $NewW[dc2] += Weights[dc1]$ 
14:      end if
15:    end for
16:  end for
17:   $Weights \leftarrow NewW$ 
18:   $DC_{db} \leftarrow top(sort(Weights), 2f + 1)$ 
19:   $l_n, DC_{rc}, DC_{cc} \leftarrow evalLat(DC_{db}, RTT, C)$ 
20: end while

```

4 SURPRISING PLACEMENT LESSONS

During the development of GPlacer, we learned some counter-intuitive lessons about data placement that exploit the diversity

and the asymmetry of the WAN links to decrease the execution and the commit latencies, hence the transaction latency. (The transaction latency L_t is sum of the execution latency L_e and the commit latency L_c .) In this Section, we explain the details of these placement lessons and their effect on the transaction latency.

4.1 Request handoff

A client executes either read or commit requests. The latency of these two requests can be abstracted as the sum of: RTT_{c-to-c} , the round-trip time between the client and the request coordinator and L_p , the time for the coordinator to process the request.

Therefore, the request latency is mainly affected by the distance between the client and the coordinator, the distance between the coordinator and the participants, and finally the number of communication rounds required between the coordinator and the participants to serve the request. Different transaction management protocols choose the coordinator based on some intuitive heuristics. In [21], Mahmoud et al. assume that the *client* is the coordinator of a transaction. In Spanner [11], the 2PC coordinator is randomly chosen from the leaders of the partitions involved in a multi-partition transaction. In [23], Nawab et al. choose the coordinator to be the closest replica to the client. However, the choice of the coordinator can drastically affect the request latency. To illustrate this effect, we provide two examples of 2PC and Paxos deployments to show that carefully choosing the coordinator can save up to 48% of the average latency.

Two-phase commit: assume there are three data partitions X , Y , and Z deployed in three AWS datacenters in SP , V , and I respectively. Now, assume a client in datacenter I wants to commit a transaction t that updates the elements $x_1 \in X$, $y_1 \in Y$, and $z_1 \in Z$. The commit latency at any coordinator equals to double the RTT between the coordinator and the furthest involved partition leader. Therefore, if the client chooses the leader of partition Z in datacenter I to be the commit coordinator, the resulting commit latency is $2 \cdot \max(RTT_{IV}, RTT_{ISP}) = 2 \cdot 185 = 370ms$. Although the time between the client and the coordinator is neglected, the latency is still high because the coordinator is relatively far from SP . However, if the client chooses the leader of partition Y in datacenter V to be the commit coordinator, the resulting commit latency is $RTT_{IV} + 2 \cdot \max(RTT_{VI}, RTT_{VSP}) = 73 + 2 \cdot 121 = 315ms$ saving around 15% of the commit latency without modifying any constraint of the original 2PC protocol. Also, when datacenter V is the 2PC coordinator, the participant at datacenter I will be notified about the commit decision after $\frac{RTT_{IV}}{2} + \max(RTT_{VI}, RTT_{VSP}) + \frac{RTT_{IV}}{2} = 36.5 + 121 + 36.5 = 194ms$. The participant at datacenter I can directly inform the client with the decision saving around 48% of the latency obtained when I is chosen to be the coordinator.

Paxos: assume there are five replicas of the database in datacenters I , V , SP , O , and C as shown in Figure 4. A client in datacenter SP wants to commit a transaction which requires to execute the two rounds of Paxos to reach a consensus about the commit decision. The latency of the two rounds of Paxos equals to double the RTT between the coordinator and the furthest replica in the closest majority to the coordinator. Therefore, if the client in SP chooses the replica in SP to be the coordinator, the resulting commit latency equals to $2 \cdot \max(RTT_{SPSP}, RTT_{SPV}, RTT_{SPO}) = 2 \cdot \max(1, 121, 182) = 2 \cdot 182 = 364ms$. However, if the client in SP , delegates the coordination to the replica in V , the resulting commit latency will be $RTT_{SPV} + 2 \cdot \max(RTT_{VV}, RTT_{VI}, RTT_{VC}) =$

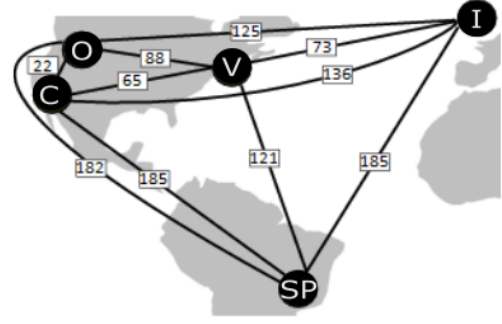


Figure 4: Five replicas of the database are deployed in datacenters I , V , SP , O , and C .

$121 + 2 \cdot 73 = 267ms$ saving around 26.6% of the commit latency obtained when SP is chosen to be the coordinator.

We presented a primitive version of the handoff idea in [34]. To generalize, for any request R from a client at datacenter A , it might be beneficial to handoff this request to a replica at datacenter B if the summation of RTT_{AB} and the time for datacenter B to serve this request L_B are less than L_A , the time to serve this request at datacenter A . In other words, request handoff from datacenter A to datacenter B is beneficial if $L_A > RTT_{AB} + L_B$. This optimization is widely applicable on different protocols and different request types.

4.2 Cover all the optimization aspects

During our SIGMOD demo [34], we ask the participants to place 5 data replicas in 5 out of the 9 datacenters shown in Figure 5. The participants are told that the data is fully replicated in the 5 chosen datacenters and the commitment protocol uses the two rounds of Paxos for isolation and replication and reads are served from the closest quorum. In addition, optimistic read and passive replica reads can be used and there is no restriction on the number of passive replicas that can be used. Finally, the workload at each site is represented by the number of blue clients at each datacenter and handoff can be used when possible.

When passive replica read is enabled and there is no restriction on the number of passive replicas assuming low contention, the commit latency contributes the most to the average transaction latency. As explained in Section 2, the commit latency is expressed as: $L_c = RTT_{c-to-c} + 2 \cdot RTT_{c-to-p}$.

Most of the demo participants tried to optimize the time to reach the commit coordinator c_c by placing the active replicas at the datacenters that have clients. Although this strategy is intuitive and reduces the time to reach c_c to zero, it does not find the placement that minimizes the overall average commit latency. This happens because the datacenters that have clients happen to be far from each other and the time to reach a quorum of participants is maximized using this strategy.

Figure 5 shows that the placement that minimizes the commit latency must consider all the optimization aspects of the commit latency. It places quorums of replicas close to each other (quorums are shown using the dotted curves) and uses handoff to handoff the commitment to replicas that can quickly form a quorum (handoff is shown using solid arrows). **Surprisingly, in this example, non of the chosen replicas are placed in datacenters that have clients.**

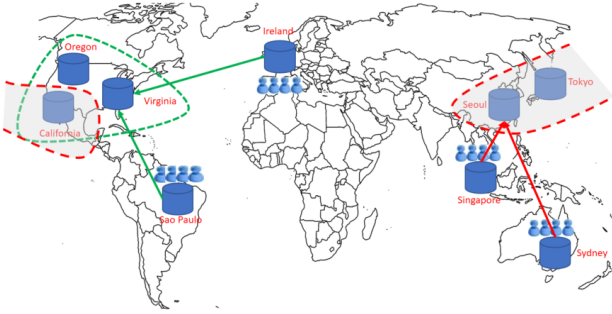


Figure 5: A placement scenario that shows the importance of considering different optimization aspects to minimize the average transaction latency.

5 EVALUATION

A performance evaluation study of the request handoff, the read optimizations, and the proposed heuristics is conducted in this section. In our study, we first evaluate the effect of the read optimizations and the request handoff on execution and commit latencies in Section 5.1. In Section 5.2, we evaluate the performance of the replica-placement heuristics introduced in Section 3.2. We compare the running time and the resulting placement latencies of these heuristics to the running time and the placement latencies of the exhaustive search algorithm in Algorithm 1.

5.1 Placement optimizations

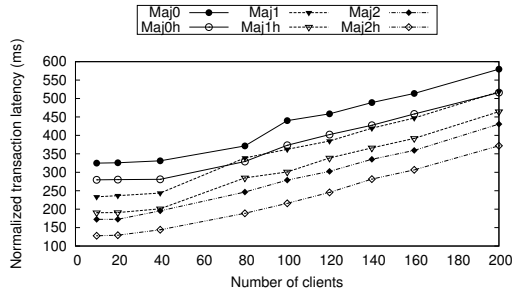
5.1.1 Experimental setup. We use the placement scenario in Figure 4 to evaluate the effect of read optimizations and request handoff on the transaction latency. Request handoff exploits the diversity of the WAN links to decrease the transaction latency and this scenario shows a good example of this diversity, $RTT_{SPC} > 8RTT_{OC}$. Amazon EC2 machines in Ireland (*I*), Virginia (*V*), São Paulo (*SP*), Oregon (*O*), and California (*C*) datacenters are leveraged as infrastructure for our experiments. Larger machines are used in datacenters *C* and *O* so that we can measure the handoff effect without causing throttling in datacenters *C* and *O*. Compute optimized machines are used because computing is the main source of contention in our experiments. We use one compute optimized (c4.large) machine with 2 vCPUs and 3.75 GB of RAM in datacenters *V*, *I*, and *SP* while we use one compute optimized (c3.4xlarge) machine with 16 vCPUs and 30 GB of RAM in datacenters *C* and *O*. We assign active replicas to servers in *C*, *O*, and *V* while we assign passive replicas to servers in *I* and *SP*. These machines use HBase [3] as the underlying persistent data store. The average RTTs observed between different datacenters are shown in Table 1. The observed RTTs are sampled over 48 hours using AWS nano machines pinging each other. The data is fully replicated in all five datacenters and an optimistic Paxos-based concurrency control protocol is used. A transaction requires two majority rounds to commit and the read-set is validated at commit time. We implemented multiple versions of the protocol based on how read requests are processed in the execution phase. *Maj0* is conservative and requires read requests to be processed from a majority of the active replicas. *Maj1* implements the optimistic read optimization and requires read requests to be processed from one active replica. *Maj2* implements the optimistic read and passive replica optimizations and processes read requests from either active or passive replica. Transaction commitment is implemented the same way in all three versions.

The commit handoff optimization is applied on all three versions and it only changes the way a commit coordinator is chosen. For this, we implemented *Maj0h*, *Maj1h*, and *Maj2h* to apply the handoff optimization on the three protocol implementations. We compare the average obtained commit and transaction latencies for all the three implementations with and without applying the handoff optimization. In addition, we compare transaction throughputs and abort rates for all three implementations.

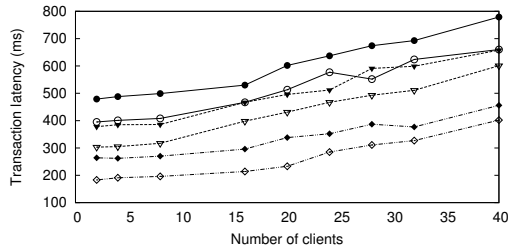
Dedicated client machines in each datacenter generate client workloads. Each client machine is configured with a read coordinator and a commit coordinator. Also, client machines execute a workload thread per client. Clients are uniformly distributed among the 5 datacenters in all the experiments unless otherwise stated. Client machines use YCSB [10] to generate workloads. Since YCSB is not designed to generate multi-record transactions, we use Transactional YCSB (T-YCSB) [12], an extended version of YCSB that generates multi-record transactions, for this purpose. T-YCSB generates transactions that consist of read and write operations on different data records followed by a commit. Each transaction is configured to have five operations. The ratio of read to write operations is 1:1 unless otherwise specified. Read and write operations choose a key from a pool of 50000 keys following a zipfian distribution. This small number of keys enables us to observe the performance of the system under contention. Each client can have only one outgoing transaction. Clients submit a new transaction as soon as they receive a decision for their outgoing transaction. Each experiment runs for 10 minutes.

5.1.2 Experimental results. Transaction latency. Active replicas are placed in only three datacenters *C*, *O*, and *V*. Therefore, a majority quorum consists of *two* active replicas. The *Maj0* implementation assumes that clients at each datacenter drive their transactions (no handoff). Also, it assumes that reads have to be processed from at least two active replicas and commits have to be accepted by and applied to at least two active replicas. *Maj0h* allows clients in *SP* to handoff their commit to *O* and clients in *I* to handoff their commits to *C*. *Maj1h* and *Maj2h* allow the same handoff plans while enabling optimistic reads in *Maj1h* and optimistic reads and passive replica reads in *Maj2h*.

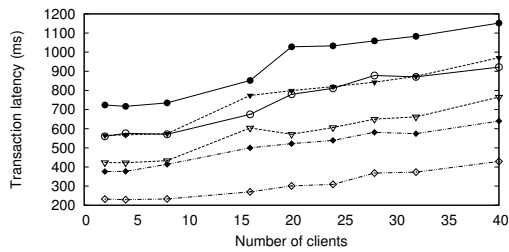
Figure 6 shows the effect of increasing the number of clients from 10 to 200 on transaction latency. As a transaction requires two round trips to a quorum of two active replicas to commit, clients in *C* and *O* have their location as an advantage that they can always achieve lower transaction latency and higher throughput than clients in other sites as long as the number of clients is equal in all the datacenters. Therefore, to measure the effect of the placement optimizations on transaction latency in isolation from the throughput, we use the normalized transaction latency as a comparison metric between different implementations. The normalized transaction latency L_{norm} is the average of the average transaction latency in all the datacenters $L_{norm} = \frac{L_C + L_O + L_V + L_I + L_{SP}}{5}$ where L_i is the average transaction latency at datacenter *i*. As shown in Figure 6a, applying read optimizations in *Maj2* significantly enhances L_{norm} by 48% compared to *Maj0*. Also, the handoff in *Maj2h* enhances L_{norm} by 26% compared to *Maj2* leading to a total enhancement of 60% compared to *Maj0*. Increasing the number of clients beyond 100 (20 at each datacenter) causes throttling in server machines. This throttling leads to an increase in the overall transaction latency and a decrease in the benefit obtained from the applied optimizations. Figures 6b and 6c shows the effect of applying read optimizations and handoff on transaction latencies at *I* and *SP*



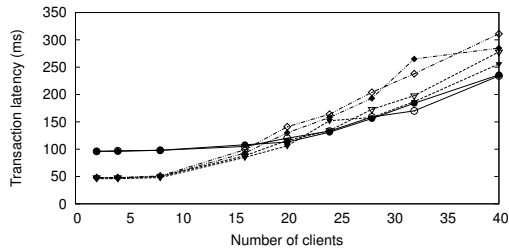
(a) Overall normalized transaction latency.



(b) Transaction latency in Ireland.



(c) Transaction latency in São Paulo.



(d) Transaction latency in California.

Figure 6: Transaction latency as number of clients increases. Figures 6a, 6b, 6c, and 6d share one plotting legend.

respectively. As shown, read optimizations and handoff together in *Maj2h* enhances transaction latency compared to *Maj0* by 62% and 68% in *I* and *SP* respectively. Also, handoff in *Maj2h* saves 30% and 38% of the transaction latency compared to *Maj2* for clients in *I* and *SP*. Figure 6d presents the effect of the placement optimizations on the transaction latency in *C*. As shown, read optimizations significantly reduce the transaction latency in *C* by 49% as reads are served locally. This applies until throttling happens. After throttling, the transaction latency in *C* increases for *Maj2* because serving reads locally in all the datacenters increases the frequency of the transactions that are ready to commit in the system causing more contention in datacenters *C* and *O*. The handoff slightly increases the transaction latency in *C* and its negative effect is negligible before the throttling happens.

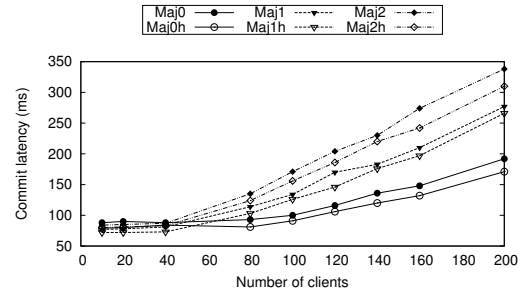
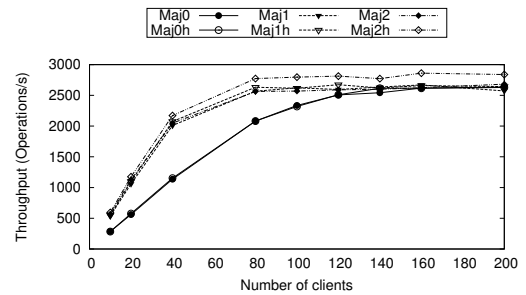
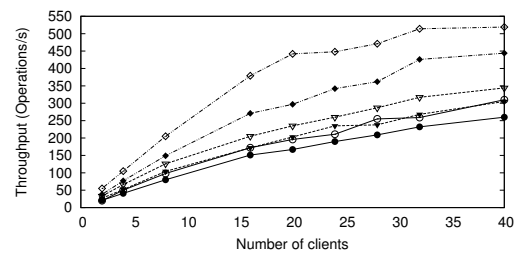


Figure 7: Overall average commit latency as number of clients increases.

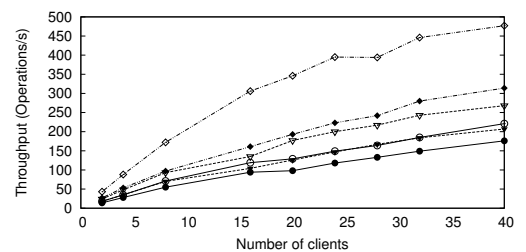
Commit latency. Figure 7 shows the effect of the placement optimizations on the overall average commit latency. While the normalized transaction latency is significantly enhanced by applying read optimizations, read optimizations negatively affect the overall commit latency. By reducing the execution phase latency, the number of active transactions that are ready to commit increases and leads to an increase in the commit latency. However, applying handoff enhances the overall average commit latency by 10 – 15% in *Maj0h*, *Maj1h*, and *Maj2h* compared to *Maj0*, *Maj1*, and *Maj2* respectively.



(a) Overall throughput.



(b) Throughput in Ireland.



(c) Throughput in São Paulo.

Figure 8: Throughput as number of clients increases. Figures 8a, 8b, and 8c share one plotting legend.

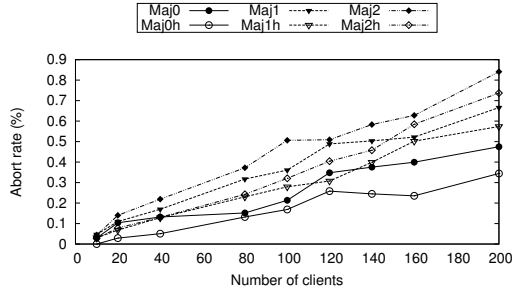


Figure 9: Overall abort rate as number of clients increases.

Throughput. The throughput, measured by number of operations per second, is presented in Figure 8. Figure 8a shows that applying read optimizations in *Maj1* and *Maj2* achieves 2x the throughput in *Maj0* until hitting the thrashing point (≥ 100 clients). After that, throughput is slightly higher in *Maj1*, *Maj2*, and *Maj1h* and about 8% higher in *Maj2h*. Throughput results in *I* and *SP* are shown in Figures 8b and 8c. These figures show a significant increase of 100% between *Maj0* and *Maj2h* in *I* and 170% between the same implementations in *SP*. Applying hand-off not only significantly benefits *I* and *SP* but also benefits the overall throughput.

Abort rate. The abort rates are shown in Figure 9. The abort rate is a result of many factors, such as the amount of contention, the number of concurrent transactions, the lifetime of a transaction, among others. As shown, the overall abort rate is below 1% for all six different implementations. However, we observed two important patterns that are worth analyzing. First, read optimizations increase the abort rate by 100% for some experiment runs. Obtaining the read-set from a local copy increases the chances of reading a stale value and hence increasing transaction aborts. However, these stale values has a small life-time as all the passive replicas are asynchronously updated. Second, handoff decreases the abort rate by 25–30% because a transaction’s lifetime is shortened by reducing the overall transaction latency and specifically the high latency transactions in *I* and *SP*.

5.2 Replica-placement heuristics

We evaluate the replica-placement heuristics in this section. This evaluation tries to answer two questions: *How fast can these heuristics find a placement? and how good is this placement compared to the optimal placement?* For that, we compare the performance and the resulting placements of the proposed heuristics in Algorithms 2 and 3 to the performance and the resulting placements of the exhaustive search in Algorithm 1 at scale. We assume that optimistic reads and passive replica reads are enabled. Therefore, we use commit latency as a comparison metric as the transaction execution latency is negligible when reads and writes are served locally and none of the replicas are overloaded with requests. The proposed heuristics and the exhaustive search programs are all implemented in Java which allows us to conduct a fair comparison. The exhaustive search algorithm evaluates all possible placement combinations and returns the placement that achieves the minimum average commit latency for a certain workload. Algorithm 2 introduces a greedy heuristic that adds one replica at a time achieving the minimum average transaction latency at each iteration. Algorithm 3 is inspired by the K-Means algorithm and it assigns initial weight to each datacenter equals to the number of clients at this datacenter. Weights are updated

based on the quorums a datacenter participates at and based on handoff.

Finding the optimal placement of five replicas within ten datacenters can be efficiently done. It requires the evaluation of only 252 different placements and the exhaustive search is sufficient in this case. However, in a more realistic setting, the number of datacenters around the globe, including edge datacenters, may easily exceed 4000 datacenters [1]. Also, it has been shown in [33] that it is economically efficient to deploy storage in datacenters of different cloud providers as non of them provides cheaper storage in all the deployment regions. To choose five datacenters out of 4000 datacenters requires to evaluate $8.5e+15$ different placements. To get a sense of the space size and the running time, we evaluated the exhaustive search algorithm and the heuristics proposed in Section 3.2 using different datasets.

First, we generate multiple datasets of datacenters *DC* distributed around the globe with randomly chosen round-trip time $0 \leq RTT \leq 500$ ms. We also make sure that the triangle inequality holds among any three datacenters such that $\forall A, B, C \in DC, RTT_{AB} + RTT_{BC} \geq RTT_{AC}$. Second, we distribute the workload around the generated datacenters with ratios between 0–10. We use the generated data as inputs to both the exhaustive search program and the placement heuristics. These experiments are run locally on an Intel Core i5-3210M CPU 2.50GHz with 8GB of RAM.

Running time. In this part of the evaluation, we answer the first question, namely *How fast can these heuristics find a placement?* Figures 10a and 10b show a running time comparison between the exhaustive search and the placement heuristics when the number of replicas are 5 and 7 respectively. As shown, the running time of the exhaustive search grows exponentially with the number of datacenters while the running time of both heuristics are negligible (< 1 second). Also, the exponential power significantly increases as the number of replicas required to be placed increases. This shows that it is infeasible to use the exhaustive search when the datacenter set size exceeds few tens.

Resulting placements. The second part of the evaluation answers the second question about the quality of the placements found by the proposed heuristics. We compare the commit latency of the resulting placement to the optimal commit latency of the resulting placement decided by the exhaustive search. Figures 11a and 11b show the relative commit latency of the heuristics compared to the optimal commit latency when the number of placed replicas are 5 and 7 respectively. In these figures, the optimal commit latency is represented by 1.0. A relative comparison of the commit latency is shown for both the placement heuristics and the best of the two heuristics as well. As shown, the best of the two heuristics is optimal in 70% of the cases and within 5%–11% of the optimal in the rest of the cases. As the running times of both heuristics is negligible, we can always run both the heuristics and choose the best placement out of the two results. Figure 11 suggests that neither heuristics beats the other in all cases.

6 GPLACER EXTENSIONS

In this section, we discuss how GPLacer can be extended to optimize the placement for leader-based protocols. In leader-based protocols, a transaction can fall into one of two categories: single-partition transactions or multi-partition transactions. Single-partition transactions span only one partition and the isolation between transactions that span this partition is managed by the

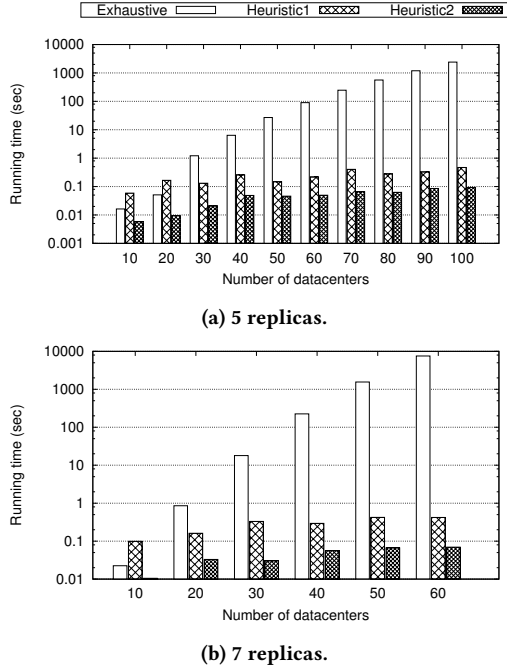


Figure 10: The running time (seconds), in log scale, of exhaustive search and placement heuristics as number of datacenters increases. Figures 10a and 10b show the running time when 5 replicas and 7 replicas are chosen respectively. Both figures share one plotting legend.

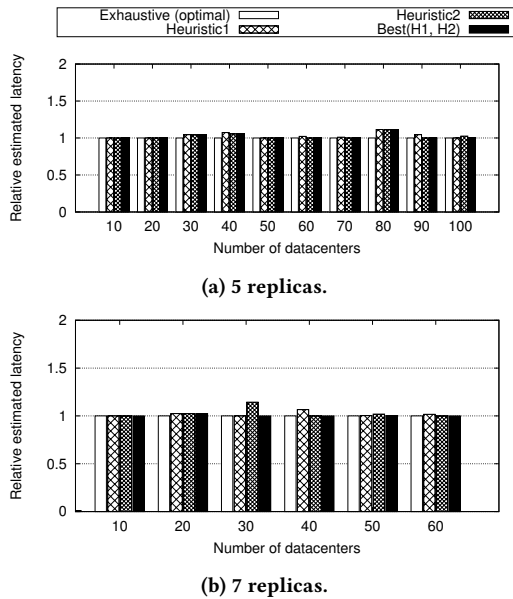


Figure 11: A comparison of the resulting commit latency of placements by exhaustive search and placement heuristics as number of datacenters increases. Figures 11a and 11b compare the estimated latency when 5 replicas and 7 replicas are chosen respectively. Both figures share one plotting legend.

leader of this partition. Multi-partition transactions span multiple partitions and typically 2PC is used between the leaders of the partitions involved in a transaction to achieve isolation. In both

categories, partition leaders replicate the updates of committed transactions to a majority quorum of their partition replicas using only the second round of Paxos.

The average transaction latency for leader-based protocols is affected by the following factors:

- The distance between the client and the partition leader
- The distance between the partition leader and its replicas
- The distance between different partition leaders involved in multi-partition transactions
- The percentage of multi-partition transactions P_{mp-txn} (how often 2PC is required to be executed)

The first two factors mainly affect single-partition transactions while the last two factors mainly affect multi-partition transactions. Finding the optimal placement for leader-based protocols can easily become impractical. Consider a database with p partitions and we want to place the leaders of these partitions on r replicas. There are r^p different placement combinations and finding the optimal placement by checking all the combinations is impractical. For example, if a database has 500 partitions and we want to place these partitions among 5 replicas. To find the optimal leader placements, 5^{500} different combinations need to be evaluated. Therefore, different heuristics are usually used to limit the search space.

6.1 Leader-placement heuristics

A solution that *considers all the optimization aspects* should adapt the placement based on the percentage of the multi-partition transactions P_{mp-txn} . Sharov et al. [28] place the leaders of all the partitions in one datacenter. Algorithm 4 implements the leader-placement heuristic introduced in [28]. It iterates over all the replicas, line 4, and evaluates the latency assuming that all the partition leaders are placed in the currently evaluated replica. The replica that achieves the minimum latency is returned. This heuristic optimizes the placement when P_{mp-txn} is high. However, when P_{mp-txn} is low, placing the leaders of all the partitions in one datacenter can hurt the performance in addition to introducing a single point of failure.

The second heuristic is to *independently* place the leaders of different partitions. For every partition, place its leader at the same datacenter where it is accessed the most. This heuristic optimizes the placement when P_{mp-txn} is low and partitions are mostly accessed from one datacenter.

Algorithm 4 Finds datacenter $dc_l \in DC$ that achieves the minimum average transaction latency assuming all the partition leaders are put together in one datacenter.

Input: $DC, RTT_{ij} \forall i, j \in DC$, and $c_i \forall i \in DC$

Output: dc_l

- 1: $dc_l \leftarrow \emptyset, l \leftarrow MaxInt$
 - 2: **for all** $j \in DC$ **do**
 - 3: $tempL \leftarrow 0$
 - 4: **for all** $i \in DC$ **do**
 - 5: $tempL+ = c_i \cdot (RTT_{ij} + q_j)$ // q_j is the time for replica j to reach its closest quorum from DC .
 - 6: **end for**
 - 7: **if** $tempL < l$ **then**
 - 8: $l \leftarrow tempL, dc_l \leftarrow j$
 - 9: **end if**
 - 10: **end for**
-

Placing all the partition leaders in one datacenter favors multi-partition transactions while independently placing them in multiple datacenters favors single-partition transactions. When the workload is a mixture of both transaction categories, both heuristics fail to optimize the placement. Therefore, we present a third heuristic that optimizes the placement when the workload is divided between the two categories. This heuristic uses GPlacer to find the set of $2f + 1$ datacenters that should host a replica DC_{db} according the workload distribution. Then it runs Algorithm 5 to independently place the leaders of each partition among the chosen replicas. The second heuristic independently places partition leaders in the universe of all datacenters DC while the third heuristic limits the placement to the set of datacenters that are chosen by GPlacer DC_{db} . In Section 6.2, we compare the resulting placements of the three heuristics.

Algorithm 5 Places the leader of each partition among the chosen replicas and closer to the clients who access this partition the most.

Input: DC_{db} , and $p_i \forall i \in DC$ and $\forall p \in P // P$ is the set of all partitions and p_i is the percentage of access for partition p from datacenter i .

Output: $\forall p \in P l_p$

- 1: **for all** $p \in P$ **do**
- 2: $i \leftarrow \max(\forall_{j \in DC} p_j)$
- 3: $l_p \leftarrow \text{nearest}(dc \in DC_{db}, i)$ // returns the nearest datacenter $dc \in DC_{db}$ to datacenter i .
- 4: **end for**

6.2 Leader-placement heuristics evaluation

We compare the expected average commit latency of the resulting leader placements using the three heuristics. The percentage of distributed multi-partition transactions is varied and the expected commit latency is calculated for the three heuristics. Before placing partition leaders in the third heuristic, we use the exhaustive search algorithm to find DC_{db} . Then, we use the heuristic to place partition leaders among the chosen replicas.

The commit latency of a single partition transaction is estimated as the RTT from the client datacenter to the partition leader datacenter plus the RTT from the partition leader datacenter to a majority of the partition replicas. The commit latency of a multi-partition transaction requires an addition 2PC between the involved partitions. In our evaluation, we assume that the 2PC added latency is negligible if all the involved partition leaders are placed together and two round-trips to all the partition leaders if they are not placed together. This assumption favors algorithm 4 over our proposed heuristic in algorithm 5.

Figure 12 shows the expected commit latency when the three heuristics are used to place partition leaders. The expected commit latency is shown in a log scale in the y-axis and the percentage of the multi-partition transaction is shown in the x-axis. In this scenario, clients are distributed among 10 datacenters and 5 datacenters host replicas for heuristic 3. A transaction has to be replicated to 3 replicas before it is committed. As heuristic 1 places all partition leaders in one datacenters, the average commit latency does not change with the percentage of multi-partition transactions. Therefore, the average estimated commit latency is a horizontal line. However, for heuristics 2 and 3, increasing the percentage of multi-partition transactions boosts the average commit latency as the cost of the 2PC between all partition leaders increases. Figure 12 suggests that heuristic 2 should be used to place

partition leaders as long as P_{mp-txn} is low (below 9%). Heuristic 3 should be used when $9\% < P_{mp-txn} < 25\%$ and heuristic 1 should be used if P_{mp-txn} is high (above 25%). Typically, the percentage of multi-partition transactions is $< 10\%$ [31, 32].

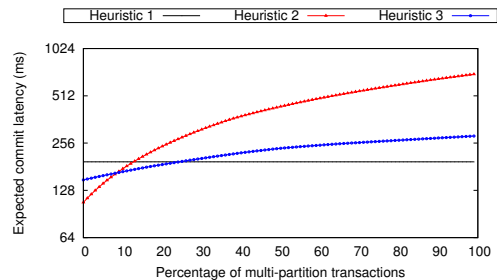


Figure 12: A commit latency comparison between leader-placement heuristics as the percentage of multi-partition transaction increases.

It is important to mention that the commit latency crossing lines between the three heuristics are different for different scenarios and estimates should be calculated a priori to decide which leader placement achieves the minimum commit latency for a given scenario. Our framework evaluates the outcomes of the three heuristics and chooses the placement that achieves the minimum latency.

7 CONCLUSION

In this paper, we address the data placement problem of geo-replicated databases with strong consistency guarantees. We present different placement optimizations to reduce transactions execution latency and commit latency. These placement optimizations are widely applied on different distributed transaction management protocols. Our evaluation shows that applying the read optimizations and the request handoff optimization could reduce transaction latency by 68% and increases throughput by 170%. To address the placement problem at scale, we propose different placement heuristics that can efficiently find sub-optimal placements within 5–10% of the optimal placements. Experiments show that these heuristics are able to scale without significantly reducing the quality of the resulting placements from the optimal placement. Finally, we discuss three partition leader placement heuristics to place partition leaders. Experiments show that non of the three heuristics is superior when the percentage of multi-partition transactions changes. Unlike in [28] which uses one heuristic to place partition leaders regardless of the percentage of multi-partition transactions, our framework switches between different heuristics when the percentage of multi-partition transactions changes.

8 ACKNOWLEDGEMENT

This work is partially funded by the NSF grant CNS-1703560.

REFERENCES

- [1] 2017. Datacenter Map. <http://www.datacentermap.com/datacenters.html/>. (2017).
- [2] 2017. GLPK: GNU Linear Programming Kit. <https://www.gnu.org/software/glpk/>. (2017).
- [3] 2017. HBase. <https://hbase.apache.org/>. (2017).
- [4] Sharad Agarwal, John Dunagan, Navendu Jain, Stefan Saroiu, Alec Wolman, and Harbinder Bhoghan. 2010. Volley: Automated Data Placement for Geo-Distributed Cloud Services.. In *NSDI*, Vol. 10. 28–0.

- [5] David F. Bacon, Nathan Bales, Nico Bruno, Brian F. Cooper, Adam Dickinson, Andrew Fikes, Campbell Fraser, Andrey Gubarev, Milind Joshi, Eugene Kogan, Alex Lloyd, Sergey Melnik, Rajesh Rao, Dave Shue, Chris Taylor, Marcel van der Holst, and Dale Woodford. 2017. Spanner: Becoming a SQL System. In *Proc. SIGMOD 2017*. 331–343.
- [6] Jason Baker, Chris Bond, James C Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. 2011. Megastore: Providing scalable, highly available storage for interactive services.. In *CIDR*, Vol. 11. 223–234.
- [7] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley.
- [8] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, and others. 2013. Tao: Facebook’s distributed data store for the social graph. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*. 49–60.
- [9] Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. 2008. PNUTS: Yahoo!’s hosted data serving platform. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1277–1288.
- [10] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 143–154.
- [11] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, and others. 2013. Spanner: Google’s globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 8.
- [12] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. 2011. Albatross: lightweight elasticity in shared storage databases for the cloud using live data migration. *Proceedings of the VLDB Endowment* 4, 8 (2011), 494–505.
- [13] Jim Gray and Leslie Lamport. 2006. Consensus on transaction commit. *ACM Transactions on Database Systems (TODS)* 31, 1 (2006), 133–160.
- [14] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems.. In *USENIX annual technical conference*, Vol. 8. 9.
- [15] Bettina Kemme and Gustavo Alonso. 2000. Don’t Be Lazy, Be Consistent: Postgres-R, A New Way to Implement Database Replication.. In *VLDB*. Citeseer, 134–143.
- [16] Tim Kraska, Gene Pang, Michael J Franklin, Samuel Madden, and Alan Fekete. 2013. MDCC: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 113–126.
- [17] Avinash Lakshman and Prashant Malik. 2009. Cassandra: structured storage system on a p2p network. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*. ACM, 5–5.
- [18] Qian Lin, Pengfei Chang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Zhengkui Wang. 2016. Towards a non-2pc transaction management in distributed database systems. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 1659–1674.
- [19] Guoxin Liu and Haiying Shen. 2016. Minimum-cost Cloud Storage Service Across Multiple Cloud Providers. In *Distributed Computing Systems (ICDCS), 2016 IEEE 36th International Conference on*. IEEE, 129–138.
- [20] Haonan Lu, Christopher Hodsdon, Khiem Ngo, Shuai Mu, and Wyatt Lloyd. 2016. The SNOW Theorem and Latency-Optimal Read-Only Transactions.. In *OSDI*. 135–150.
- [21] Hatem Mahmoud, Faisal Nawab, Alexander Pucher, Divyakant Agrawal, and Amr El Abbadi. 2013. Low-latency multi-datacenter databases using replicated commit. *Proceedings of the VLDB Endowment* 6, 9 (2013), 661–672.
- [22] Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. 2013. Message Futures: Fast Commitment of Transactions in Multi-datacenter Environments.. In *CIDR*.
- [23] Faisal Nawab, Vaibhav Arora, Divyakant Agrawal, and Amr El Abbadi. 2015. Minimizing commit latency of transactions in geo-replicated data stores. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 1279–1294.
- [24] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, and others. 2013. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. 385–398.
- [25] Stacy Patterson and others. 2012. Serializability, not Serial: Concurrency Control and Availability in Multi-Datacenter Datastores. *PVLDB* (2012).
- [26] Fan Ping, Jeong-Hyon Hwang, XiaoHu Li, Chris McConnell, and Rohini Vabalarreddy. 2011. Wide area placement of data replicas for fast and highly available data access. In *Proceedings of the fourth international workshop on Data-intensive distributed computing*. ACM, 1–8.
- [27] Moinuddin K Qureshi, Sudhanva Gurumurthi, and Bipin Rajendran. 2011. Phase change memory: From devices to systems. *Synthesis Lectures on Computer Architecture* 6, 4 (2011), 1–134.
- [28] Artyom Sharov, Alexander Shraer, Arif Merchant, and Murray Stokely. 2015. Take me to your leader!: online optimization of distributed storage configurations. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1490–1501.
- [29] Jeff Shute, Mircea Oancea, Stephan Ellner, Ben Handy, Eric Rollins, Bart Samwel, Radek Vingralek, Chad Whipkey, Xin Chen, Beat Jegerlehner, Kyle Littlefield, and Phoenix Tong. 2012. F1 - The Fault-Tolerant Distributed RDBMS Supporting Google’s Ad Business. In *SIGMOD*. Talk given at SIGMOD 2012.
- [30] Michael Stonebraker. 2010. Why Enterprises Are Uninterested in NoSQL. <http://cacm.acm.org/blogs/blog-cacm/99512-why-enterprises-are-uninterested-in-nosql/fulltext/>. (2010).
- [31] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J Elmore, Ashraf Aboulmaga, Andrew Pavlo, and Michael Stonebraker. 2014. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proceedings of the VLDB Endowment* 8, 3 (2014), 245–256.
- [32] Alexander Thomson and others. 2012. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD*.
- [33] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V Madhyastha. 2013. SPANStore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 292–308.
- [34] Victor Zakhary, Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. 2016. Db-risk: The game of global database placement. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2185–2188.

SlickDeque: High Throughput and Low Latency Incremental Sliding-Window Aggregation

Anatoli U. Shein
Dept. of Computer Science
University of Pittsburgh
aus@cs.pitt.edu

Panos K. Chrysanthis
Dept. of Computer Science
University of Pittsburgh
panos@cs.pitt.edu

Alexandros Labrinidis
Dept. of Computer Science
University of Pittsburgh
labrinid@cs.pitt.edu

ABSTRACT

Online analytics, in most advanced scientific and business applications, rely heavily on the efficient execution of large numbers of Aggregate Continuous Queries (ACQs). Incremental sliding-window computation is used in the state-of-the-art ACQ processing algorithms (*FlatFIT*, *TwoStacks*, and *DABA*) to avoid the re-evaluation of the aggregate value of the window from scratch on every update. *FlatFIT* and *TwoStacks* aim to increase throughput, and *DABA* to minimize latency, while all process invertible and non-invertible aggregates uniformly. In this paper, we propose a novel algorithm, *SlickDeque*, that distinguishes the execution between invertible and non-invertible aggregates and offers better throughput and latency for both types. In addition, our method requires less memory and efficiently supports *multi-ACQ* processing. We theoretically show the time and space complexity advantages of *SlickDeque* and experimentally validate them using a real workload. Specifically, our approach maintains 283% lower latency spikes on average while achieving up to 19% throughput improvement in a single query environment and up to 345% improvement in a multi-query environment over the state-of-the-art approaches along with requiring up to 5 times less memory.

1 INTRODUCTION

Motivation Data stream processing has gained momentum in many applications that require quick responses based on incoming high velocity data flows. A representative example is a stock market application, where multiple clients monitor the price fluctuations of the stocks. In this setting, a system needs to be able to efficiently answer analytical queries (e.g., average stock revenue, profit margin per stock, etc.) for different clients, each one with (possibly) different timing requirements. Efficient data stream processing is also important in monitoring applications in the fields of health care, science, social media, and network control.

Data Stream Management Systems (DSMS) [1–3, 22, 30] have been proposed as the most suitable systems for handling such data flows on-the-fly and in real time. In a DSMS, clients register their analytical queries on incoming data streams. These queries continuously aggregate streaming data, and as such they are called Aggregate Continuous Queries (ACQs). ACQs are typically associated with a *range* (r) and a *slide* (s) (also referred to as *window* and *shift* [15]), which can be either count or time-based. A slide denotes the period at which an ACQ updates its answer; a range is the window for which the statistics are calculated.

An ACQ requires the DSMS to keep state over time while performing aggregations. Normally, DSMSs only keep the window of the most recent data, and produce the answers by running aggregate queries over it. It has been shown that in sliding-window

stream processing, it is beneficial to use *incremental evaluation*, which involves storing and reusing calculations performed over the unchanged parts of the window, rather than performing the re-evaluation of the entire window after each update [10, 20]. Incremental evaluation typically runs partial aggregations on the data and produces the answer by performing the final aggregation over the partial results [18, 19].

Problem Statement Handling of aggregate operations that are both *invertible* and *non-invertible* proved to be essential in domains such as finance and science. *Invertible* operations include Sum, Product, Count, Average, and Standard Deviation, while *non-invertible* operations include Max, Min, Range, Alphabetical Max (for strings), ArgMax of Cosine, and ArgMin of x^2 . It was shown previously that *invertible* operations can be processed efficiently by maintaining a running Sum (or other aggregation), and invoking the inverse operation (such as Subtract) on every expiring tuple, however *non-invertible* operations require more effort to be processed efficiently and remain a challenge.

The current state-of-the-art solutions for processing ACQs, *FlatFIT* [26] and *TwoStacks* [28], aim to increase throughput and *DABA* [28], to minimize latency. These solutions process invertible and non-invertible aggregates uniformly, which negatively affects their performance with increasing workloads. To address the aforementioned shortcomings, in this paper we propose a novel solution named *SlickDeque*, which handles aggregate operations differently based on their invertibility property. The invertible operations are processed using *SlickDeque* (Inv), our new modified *Panes* (Inv) approach, while non-invertible ACQs are processed with *SlickDeque* (Non-Inv), our novel deque-based algorithm that intelligently maintains and utilizes intermediate partial aggregates allowing a greater level of reuse of previously calculated results. The separation based on invertibility leads to exceptional throughput and latency for both invertible and non-invertible operations in systems with heavy workloads.

We consider also *multi-query*, *multi-tenant* environments, where large numbers of ACQs with different ranges and slides operate on the same data stream, calculating similar aggregations.

Contributions We make the following contributions:

- We propose a novel solution for processing ACQs, *SlickDeque*, which processes invertible and non-invertible operations differently. *SlickDeque* is applicable for both single query and multi-query environments. (Section 3)
- We theoretically evaluate *SlickDeque* and show that it achieves better time and space complexities compared to the state-of-the-art *FlatFIT*, *TwoStacks*, and *DABA* solutions. To our knowledge, there are no prior algorithms that can achieve the same time and space complexities without loss of query generality in terms of supported aggregate operations. (Section 4)
- We experimentally evaluate *SlickDeque* based on a real dataset and show that it significantly outperforms state-of-the-art techniques in all tested scenarios by increasing the ACQ

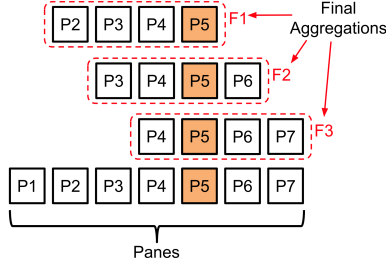


Figure 1: Panes Technique

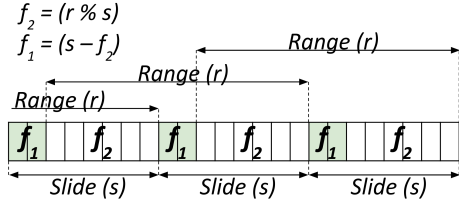


Figure 2: Paired Window Technique

throughput by up to 19% in a single query environment and by up to 345% in a multi-query environment, while maintaining 283% lower latency spikes on average and reducing memory consumption by up to 5 times. We also show that our approach becomes superior to the state-of-the-art approaches starting at window sizes as small as eight tuples with its benefits increasing rapidly, making *SlickDeque* widely applicable for processing *ACQs* in a variety of *DSMSs*. (Section 5)

2 BACKGROUND & RELATED WORK

In this section we briefly review the underlying concepts of our work, which are the incremental sliding-window computation techniques. These could be broadly divided into *partial aggregation* and *final aggregation*. We also review other related work.

2.1 Partial aggregation

Partial aggregation can be thought of as the buffering of partial results until the query result needs to be returned by the final aggregation. Since partial aggregation allows some buffering before the result needs to be processed by a more expensive final aggregator and each buffered partial can be reused multiple times as part of final aggregations, the use of the CPU and memory resources to maintain the partials can be amortized. The following techniques aiming to reduce the number of partials were proposed for partial aggregations.

Panes [19] was proposed as the first partial aggregation technique for processing *ACQs* efficiently. The idea behind it is to partition the incoming datastream into “*panes*” (we refer to them as *partials*), and maintain just one aggregate value for each partial. This way every incoming tuple will affect the aggregate value for just the current partial, and when the whole aggregate is due to be reported, the answer is assembled by performing the final aggregation over all the partials in the current window. Therefore, each new partial will be reused multiple times for different final aggregations. For example, in Fig. 1 partial *P5* is used 3 times as part of the final aggregations *F1*, *F2*, and *F3*.

Paired Window technique, or simply *Pairs* [18], was introduced to reduce by a factor of 2 the number of partials in a window in cases where the range is not divisible by the slide, reducing the

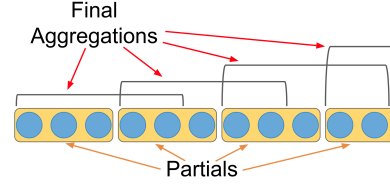


Figure 3: Cutty-slicing Technique

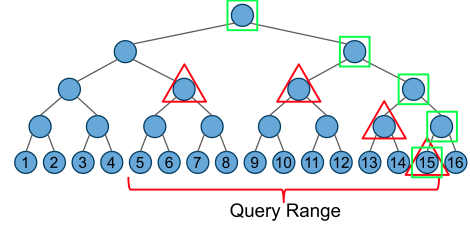


Figure 4: FlatFAT Technique

memory consumption and accelerating the final aggregations. As illustrated in Fig. 2, two fragment lengths are used, f_1 and f_2 , where $f_2 = range \% slide$ and $f_1 = slide - f_2$.

Cutty-slicing was proposed as part of the *Cutty* optimizer [8], and it starts each new partial only at positions that signify the beginning of new windows. This way the final aggregation can execute in the middle of the partial aggregation calculation by accessing the current value in the partial (Fig. 3). This reduces the number of partials per window by a factor of two compared to *Pairs* but it comes at a cost: additional punctuations have to be sent over the data stream to the execution module to indicate the beginnings of the new partials, which reduces the effective bandwidth of the stream and can slow down the system, especially if the workload includes a large number of queries with small windows.

2.2 Final Aggregation

The goal of final aggregation is to produce the result of a query by utilizing the partials. Initially it was performed by simply iterating over them and constructing the answer [18, 19]. For example the *Panes* technique (which we consider *Naive* in this work) in Fig. 1 performs a final aggregation *F1* by iterating over partials *P2*, *P3*, *P4*, and *P5*. Naturally, such a solution quickly became outdated due to the increasing workloads that created bottlenecks in the final aggregator. To improve this, several final aggregation techniques have been proposed [5, 21, 26–29, 31].

Panes (Inv) [19] (or *Panes* for Invertible (Differential) Aggregate Queries) was proposed to efficiently process invertible aggregates, and it works by maintaining a running aggregate (e.g. running Sum), and invoking the inverse operation (e.g. Subtract) on every expiring tuple. This algorithm (with minor differences) was also proposed as *R-Int* [5] and *Subtract-on-Evict* [28]. In this paper we extend this approach into *SlickDeque (Inv)*, which can do multi-query processing by maintaining a running aggregate for each query with a distinct range registered on the data stream.

Despite being very effective, *Panes (Inv)* is only applicable for invertible operations. In order to allow greater generality in query processing, the following techniques have been introduced.

FlatFAT [29] (or Flat Fixed-sized Aggregator) is a final aggregation approach which stores tuples in a pre-allocated pointer-less



Figure 5: B-Int Technique

tree-based data structure (Fig. 4), and was later extended [8] to allow partial aggregation and multi-query processing by allowing to store partial aggregates as tree leaves. Each internal node of the tree contains an aggregate of its two children. New partials are inserted into the leaves of the binary tree left-to-right. The leaves form a circular array, meaning that after inserting a value to the rightmost leaf, the next insert will go into the leftmost one. Each insert triggers the update procedure, which is performed by walking the tree bottom-up and updating all internal nodes. An example of an update operation on leaf 15 is illustrated with green squares in Fig. 4. The look-up of the answer in *FlatFAT* is performed by returning the root node value if a query requires the result for the maximum window, or by aggregating a minimum set of internal nodes that covers the required range of leaves. The example of answering a query with a range of 11 partials starting from leaf 15 is shown with red triangles in Fig. 4. **B-Int**[5] (or Base Intervals) is another final aggregation technique that uses a multi-level data structure that consists of dyadic intervals of different lengths. On the first level, the intervals are of a length of one partial, on the next level the interval length is two partials, on the third level the length is four partials, and so on. The top level has just one interval of the maximum supported range length. The whole data structure is organized in a circular fashion, so that the rightmost interval on any level is followed by the leftmost interval from the same level (Fig. 5). Similarly to *FlatFAT*, when producing the final aggregate, *B-Int* determines the minimum number of intervals needed to represent the desired range, and aggregates them. For example, in Fig. 5 *B-Int* aggregates all intervals marked with color to get the answer for the specified query range. Another tree-like approach similar to *FlatFAT* and *B-Int* is [6].

FlatFIT [26] (or Flat and Fast Index Traverser) was proposed with a goal of increasing the throughput of *ACQ* processing. *FlatFIT* achieves acceleration by dynamically storing the intermediate results and their corresponding pointers, which indicate how far ahead *FlatFIT* can skip in its calculation. It uses two circular arrays, *Pointers* and *Partials*, interconnected with their indices and a stack, *Positions*, for keeping indices that are currently processed. The *FlatFIT* algorithm is applicable in a multi-query environment, where it achieves a high throughput by allowing additional partial result reuse between all *ACQs* on the stream.

TwoStacks [28] was shown to also achieve a high throughput by using an old trick from functional programming to implement a queue with two stacks, *F* (front) and *B* (back), where all insertions push a value, *val*, and an aggregation, *agg*, of everything below it onto *B*, and evictions pop from *F*. When *F* is empty, the algorithm flips *B* onto *F*, making it a calculation heavy step that introduces latency spikes to processing. To produce the final aggregation, the tops of both the *F* and *B* stacks are aggregated.

DABA [28] (or De-Amortized Bankers Algorithm) was proposed as an alternative to *TwoStacks* that reduces the latency spikes

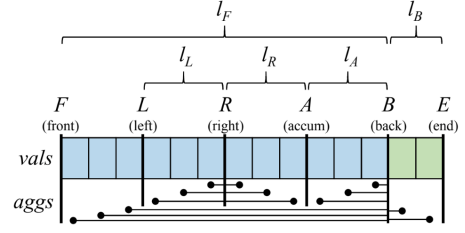


Figure 6: DABA Technique

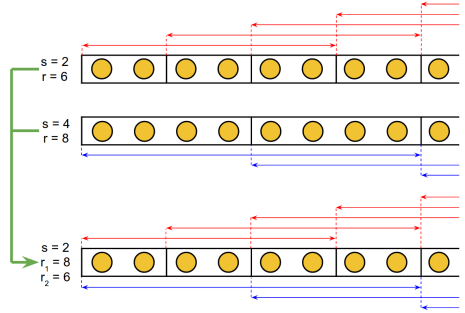


Figure 7: Shared Processing

while maintaining high throughput. The algorithm uses a principle of the Functional Okasaki Aggregator to de-amortize the *TwoStacks* algorithm. *DABA* uses two queues, *vals* and *aggs*, as shown in Fig. 6 implemented as chunked-array queues with six ordered pointers which make up the *F* and *B* stacks similarly to *TwoStacks*. However after each insertion and eviction event, a function *fixup* is called which re-balances the pointers and fixes the consistency of the *aggs* queue.

Currently, neither *TwoStacks* nor *DABA* are known to support multi-query execution as opposed to the other above algorithms.

2.3 Shared Processing of ACQs

Since the *ACQs* are executed periodically (unlike one-shot queries), several processing schemes, as well as *ACQ* optimizers, take advantage of the shared processing of *ACQs* [8, 14, 18], which reduces the long-term overall processing costs by sharing partial results. To show the benefits of sharing in such scenarios, consider the following example:

Example 1 (Fig. 7) Assume two *ACQs* monitor Max stock value over the same data stream. The first *ACQ* has a slide of 2 tuples and a range of 6 tuples, the second one has a slide of 4 tuples and a range of 8 tuples. That is, the first *ACQ* is computing partial aggregates every 2 tuples, and the second is computing the same partial aggregates every 4 tuples. Clearly, the calculation producing partial aggregates only needs to be performed once every 2 tuples, and both *ACQs* can use these partial aggregates for their corresponding final aggregations. The first *ACQ* will then run each final aggregation over the last three partial aggregates, and the second *ACQ* will run each final aggregation over the last 4 partial aggregates. ■

Partial results sharing is applicable for all matching aggregate operations, such as Max, Product, Sum, etc. and for different but compatible aggregate operations, for example Sum, Count and Average can share results by treating Average as $\frac{\text{sum}}{\text{count}}$.

To determine how many partial aggregations are needed after combining *n* *ACQs* into a shared execution plan, we first find the

length of the new composite slide, which is the *Least Common Multiple (LCM)* of the slides of the combined ACQs (in Example 1 it is four). Each slide is then repeated $LCM/slide$ times to fit the length of the composite slide, and all slide multiples are marked within the composite slide as *edges*. If slides consist of several fragments due to the partial aggregation, all fragments are also marked within the composite slide as edges. The more common edges are present in the composite slide, the more partial aggregations can be shared.

In this work we combine all compatible ACQs into one shared plan to achieve maximum sharing, which, in a general case, provides the most computational resource savings. Although, in specific cases it was shown that aiming for maximum sharing is not always beneficial [13, 14, 24, 25].

2.4 Other Related Work

Work similar to sliding-window aggregation exists in *Temporal Database Systems*, which store the entire stream of tuples and allow aggregations over any continuous segments of the stream, which are called *Historical Windows*. In contrast, *DSMSs* generally support windows that end at or near the most recent results are referred to as *Suffix Windows* in Temporal Databases. In Temporal Databases, Red-black trees [17, 21], SB-trees, B-trees [31], and Skylines [23] are used for aggregations. Due to the tree-based natures of these algorithms their update complexities are $O(\log(s))$, where s is the size of the entire stream history.

Several approximate calculation approaches were proposed to save time and space by giving up accuracy [4, 7, 9, 11]. Our approach focuses solely on computing *exact* answers since it is crucial for many applications (e.g., financial, medical, etc.).

3 SLICKDEQUE

In this section we describe our new algorithm, *SlickDeque*, that significantly speeds up the final aggregation calculations in a sliding-window environment by employing different processing schemes for invertible and non-invertible aggregations.

3.1 Algebraic Properties and Assumptions

One of the important metrics that allows the evaluation of the difficulty of incremental evaluation of a particular query is the algebraic properties of the underlying aggregate operation. Based on classification from [12], all aggregate operations are divided into three broad categories: *distributive*, *algebraic*, and *holistic*.

- **Distributive** aggregation means that the aggregation for the set S can be computed from two of the same aggregations of subsets S_1 and S_2 , where subsets S_1 and S_2 were constructed by splitting S in two. For example, if we have a set of 10 numbers and the Sum of the first 7 is 20, and the Sum of the 3 remaining is 15, then we can get the Sum of all 10 numbers by adding 20 and 15. Therefore, Sum is a distributive aggregation.
- **Algebraic** aggregation means that the aggregation can be computed from a number of distributive aggregations, e.g., Average, which is calculated from Sum and Count. The list of common distributive aggregations includes Count, Sum, Sum of Squares, Product, and Max. By combining these distributive aggregations we can calculate some commonly used algebraic aggregations such as: Average (Count and Sum), Standard Deviation (Sum of Squares, Sum, and Count), Geometric Mean (Product and Count), and Range (Max and Min).

- **Holistic** aggregations are neither distributive nor algebraic, e.g., Median, Top-K, Quantile, Collect Distinct. Holistic aggregations are out of the scope for this work since they require specifically tailored algorithms which cannot be generalized.

In this paper we will focus on optimizing the distributive aggregations; calculating the algebraic aggregations follows trivially. Distributive aggregations can be further classified by their mathematical properties: *associativity*, *invertibility*, and *commutativity*. Below we provide brief definitions of these properties.

- An operation \oplus is *associative* if $x \oplus (y \oplus z) = (x \oplus y) \oplus z$ is true for all x, y, z .
- An operation \oplus is *invertible* if there exists an operation \ominus such that $(x \oplus y) \ominus y = x$ for all x, y , and \ominus is feasibly inexpensive.
 - Note: if operation \oplus is *non-invertible*, then $x \oplus y = z$, where $z \in \{x, y\}$. This is only true for **non-holistic** operations (which we target in this work).
- An operation \oplus is *commutative* if $x \oplus y = y \oplus x$ is true for all x, y .

Query Operation Assumptions In terms of query operation generality, our proposed approach, *SlickDeque*, is no different from the state-of-the-art approaches, which all support non-invertible and non-commutative operations while requiring the operations to be associative. In general, all operations that can be executed on a window of values are associative. The common non-associative operations such as subtraction ($x - y - z$), division ($x/y/z$), exponentiation (x^{y^z}), and some binary operations such as *NAND* and *NOR*, are generally impractical when executed on sets of values larger than two. The difference of our proposed *SlickDeque* approach is that it has separate processing algorithms for the invertible operations (e.g., Sum, Product, Count, etc.) and non-invertible operations (e.g., Max, Min, Range, Alphabetical Max (for strings), ArgMax of Cosine, ArgMin of x^2 , etc.), which allows us accelerated processing of both.

Window Structure Assumptions In *non-FIFO* window structures, the events of insertion and expiration are not synchronized, which can cause window overflow situations when there are not enough expiring tuples (or partial aggregates) to make room in the window for the insertions. All of the compared approaches, including ours, are able to handle such cases by performing dynamic resize operations. However in this paper we are focusing on the *FIFO* window environment which is the most common way to processing sliding-window aggregations in practice.

Arrival Order Assumptions Similarly, all of the aforementioned algorithms allow updates on multiple partial aggregates already stored within the window. However in this paper we focus on the classic streaming scenario when all new partial aggregates are processed by the final aggregator one-by-one as they become available. In such settings the arriving tuples have to be *in-order* or slightly *out-of-order*. As long as the *out-of-order* tuples are within the same partial aggregation, the final result will not be affected. If, however, some tuples fall outside of their partial, inconsistencies in the final result may arise. The mechanism that all systems uses to cope with such extreme situations is outside of the scope of this paper.

3.2 The SlickDeque Algorithm

In this subsection we provide the algorithm and implementation details for our approach followed by the clarifying examples. We

Algorithm 1 SlickDeque (Inv) Pseudocode

```
1: Input: A set of aggregate continuous queries  $Q$ , invertible aggregate
   operation  $\oplus$ , the initial value for  $\oplus$   $initVal$ , the inverse operation  $\ominus$ ,
   and partial aggregation technique PAT
2: Output: Continuous answers to queries in  $Q$  according to their
   specifications.
3:           Phase 1 (Preparation)
4: sharedPlan = buildSharedPlan( $Q$ , PAT)
5: wSize = sharedPlan.wSize
6: partials = new array[wSize]
7: answers = new map(queryRange  $\rightarrow$  answer)
8: for  $i=0$  to wSize do
9:   partials[ $i$ ] = initVal
10: end for
11: for each query  $q \in Q$  do
12:   answers.insert( $q$ .range, initVal)
13: end for
14: currPos = 0
15:           Phase 2 (Execution)
16: while results are expected do
17:   length = sharedPlan.getNextPartialsLength()
18:   newPartial = partialAggregator.aggregate(length, PAT)
19:   for each ( $qR \rightarrow ans$ ) pair in answers do
20:     startPos = currPos -  $qR$ 
21:     if startPos < 0 then
22:       startPos += wSize
23:     end if
24:     ans = ans  $\oplus$  newPartial  $\ominus$  partials[startPos]
25:   end for
26:   queriesToAnswer = sharedPlan.getNextSetOfQueries()
27:   for each query  $q$  in queriesToAnswer do
28:     send answers.getVal( $q$ .range) as answer to  $q$ 
29:   end for
30:   partials[currPos] = newPartial
31:   currPos++
32:   if currPos == wSize then
33:     currPos = 0
34:   end if
35: end while
```

break down our algorithm description based on invertibility of the aggregate operator.

SlickDeque for Invertible Aggregates

For processing invertible aggregates we propose *SlickDeque* (Inv), a modified *Panes* (Inv) extended for processing multiple ACQs. Pseudocode for it is depicted in Algorithm 1. The algorithm consists of two major phases: *Preparation* and *Execution*.

The Preparation Phase given a set of queries, Q , and one of the partial aggregation techniques (PAT) discussed in Section 2.1 (e.g., *Pairs*) as an input, *SlickDeque* (Inv) builds a shared execution plan by executing the *buildSharedPlan* function (line 4). The *sharedPlan* is constructed as discussed in Section 2.1, and includes a full list of partials (or edges) augmented with their lengths and lists of queries to be evaluated for each partial. The *buildSharedPlan* function identifies the query with the longest range in terms of the number of partials, and saves the range as the member *wSize* of the *sharedPlan* (line 5). *wSize* signifies the necessary window length needed to process all input queries.

After generating the *sharedPlan*, *SlickDeque* (Inv) initializes its data structures: a circular array, *partials*, (line 6) and a map, *answers*, (line 7). The *partials* array is initialized to a length equal to *wSize*, and is used to store partial aggregates. The *answers* map maintains the mappings of all queries with unique ranges to

their current answers. Queries operating over the same range can share results even if they have different slides. Both the *partials* array and the values of the *answers* map are initialized (lines 8-13) with the initial value for the operation \oplus , *initVal*, supplied as input. For example, *initVal* is $-\infty$ for the Max operation.

The *currPos* variable signifies the current position within the *partials* array (line 14). It starts at 0 initially and increases to *wSize* - 1 during execution, after which it wraps back to 0. The arriving partial aggregates will be inserted into the *partials* array always at the *currPos*.

The Execution Phase is implemented as a loop that continuously returns all query results while they are expected. At the beginning of the loop (lines 17-18), *SlickDeque* (Inv) gets the next partial's length from the *sharedPlan*, and passes it to the *newPartial Aggregator* which uses the provided PAT technique to produce the *newPartial* value.

Next, *SlickDeque* (Inv) loops over all range-to-answer mappings ($qR \rightarrow ans$) in the *answers* map (lines 19-25). The loop starts by identifying the start position, *startPos*, for each mapping within the *partials* array from which the values need to be aggregated. *startPos* is identified by rewinding *currPos* back by query range, *qR*, length.

Since *SlickDeque* (Inv) only works for the invertible queries, it utilizes both the aggregate operation \oplus (e.g., Sum if query is seeking Sum), and an inverse operation \ominus (e.g., Subtract if the original operation is Sum). This way each answer, *ans*, is updated by executing the aggregate operation \oplus with the newly calculated *newPartial* value and the inverse operation \ominus with expiring *partials[startPos]* value (line 24).

Next, the answers to all queries scheduled at the current position need to be produced (lines 26-29). After receiving the *queriesToAnswer* (a subset of Q) from the *sharedPlan*, *SlickDeque* (Inv) loops over them while sending back the corresponding answers pulled from the *answers* map. Then, the *Partial* value is inserted into the circular *partials* array at *currPos*, and *currPos* is moved one position forward (lines 30-34).

The following Example 2 (illustrated in Fig. 8) should clarify the above algorithm. In order to make the explanation more intuitive we execute the two queries, Q_1 and Q_2 , on the same incoming datastream using two algorithms: *Naive* and *SlickDeque* (Inv), and we illustrate each step of their calculations side-by-side.

Example 2 Assume we have queries Q_1 and Q_2 , which are seeking the Sum over the ranges of 3 and 5 tuples, respectively, both with a slide of 1 tuple. The slide size is set to one tuple in this example for simplicity, which means that there is no partial aggregation and the answers to both queries need to be calculated after every new tuple arrival. Since the range of Q_2 is 5, which is greater than the range of Q_1 , and the slides of Q_1 and Q_2 are the same, the shared execution plan has a *wSize* of 5 tuples.

Both *Naive* and *SlickDeque* (Inv) algorithms use the *partials* array in order to maintain incoming partial aggregates (in this case just tuples). The difference is that *Naive* produces answers to queries by iterating over this array, while *SlickDeque* (Inv) utilizes the additional *answers* map (Introduced above).

In the *partials* array we mark the positions that have been modified by the algorithm in each step. The current position (*currPos*) at each step is bolded in Fig. 8 for convenience. The tuples enter the system in the order: 6, 5, 0, 1, 3, 4, 2, 7.

After the initialization in Step 0, in Step 1 the first tuple, 6, arrives. Both algorithms store the new tuple at the *currPos* in the *partials* array, and *Naive* iterates over indexes 3, 4, and 0 in order

Step	Naive	SlickDeque (Inv)	Answer														
0	partials: <table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	0	1	2	3	4	0	0	0	0	0	answers: <table border="1"><tr><td>3</td><td>5</td></tr><tr><td>0</td><td>0</td></tr></table>	3	5	0	0	Q1 Q2 n/a n/a
0	1	2	3	4													
0	0	0	0	0													
3	5																
0	0																
1	partials: <table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>6</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	0	1	2	3	4	6	0	0	0	0	answers: <table border="1"><tr><td>3</td><td>5</td></tr><tr><td>6</td><td>6</td></tr></table>	3	5	6	6	6 6
0	1	2	3	4													
6	0	0	0	0													
3	5																
6	6																
2	partials: <table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>6</td><td>5</td><td>0</td><td>0</td><td>0</td></tr></table>	0	1	2	3	4	6	5	0	0	0	answers: <table border="1"><tr><td>3</td><td>5</td></tr><tr><td>11</td><td>11</td></tr></table>	3	5	11	11	11 11
0	1	2	3	4													
6	5	0	0	0													
3	5																
11	11																
3	partials: <table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>6</td><td>5</td><td>0</td><td>0</td><td>0</td></tr></table>	0	1	2	3	4	6	5	0	0	0	answers: <table border="1"><tr><td>3</td><td>5</td></tr><tr><td>11</td><td>11</td></tr></table>	3	5	11	11	11 11
0	1	2	3	4													
6	5	0	0	0													
3	5																
11	11																
4	partials: <table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>6</td><td>5</td><td>0</td><td>1</td><td>0</td></tr></table>	0	1	2	3	4	6	5	0	1	0	answers: <table border="1"><tr><td>3</td><td>5</td></tr><tr><td>6</td><td>12</td></tr></table>	3	5	6	12	6 12
0	1	2	3	4													
6	5	0	1	0													
3	5																
6	12																
5	partials: <table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>6</td><td>5</td><td>0</td><td>1</td><td>3</td></tr></table>	0	1	2	3	4	6	5	0	1	3	answers: <table border="1"><tr><td>3</td><td>5</td></tr><tr><td>4</td><td>15</td></tr></table>	3	5	4	15	4 15
0	1	2	3	4													
6	5	0	1	3													
3	5																
4	15																
6	partials: <table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>4</td><td>5</td><td>0</td><td>1</td><td>3</td></tr></table>	0	1	2	3	4	4	5	0	1	3	answers: <table border="1"><tr><td>3</td><td>5</td></tr><tr><td>8</td><td>13</td></tr></table>	3	5	8	13	8 13
0	1	2	3	4													
4	5	0	1	3													
3	5																
8	13																
7	partials: <table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>4</td><td>2</td><td>0</td><td>1</td><td>3</td></tr></table>	0	1	2	3	4	4	2	0	1	3	answers: <table border="1"><tr><td>3</td><td>5</td></tr><tr><td>9</td><td>10</td></tr></table>	3	5	9	10	9 10
0	1	2	3	4													
4	2	0	1	3													
3	5																
9	10																
8	partials: <table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>4</td><td>2</td><td>7</td><td>1</td><td>3</td></tr></table>	0	1	2	3	4	4	2	7	1	3	answers: <table border="1"><tr><td>3</td><td>5</td></tr><tr><td>13</td><td>17</td></tr></table>	3	5	13	17	13 17
0	1	2	3	4													
4	2	7	1	3													
3	5																
13	17																

Figure 8: Example 2 processing of invertible aggregate queries Q1 and Q2 using Naive and SlickDeque (Inv) algorithms.

to answer $Q1$, and iterates over the entire array to answer $Q2$. Both answers in this case are 6.

SlickDeque (Inv) on the other hand in step 1 just updates all answers in the *answers* map by executing the operation \oplus (in this example it is Sum) with the newly arrived tuple 6 and the inverse operation \ominus (in this example it is Subtract) with values at indexes 2 and 0 in the *partials* array, which both are zeros. The updated answers are stored in the *answers* map.

In Step 2, the new partial, 5, arrives, and *Naive* iterates again over the past 3 tuples to answer $Q1$ and over the whole window to answer $Q2$, and sums up all of the values that were visited. The *SlickDeque* (Inv) algorithm on the other hand, is able to provide answers to both queries with just two operations each. It adds 5 and subtracts 0 from both answers in the map, making both 11.

Skipping ahead, in Step 4 *SlickDeque* (Inv) adds the new tuple, 1, to both answers, subtracts 6 from the answer to $Q1$ (since it is now out of range of $Q1$), and then subtracts 0 from the answer to $Q2$ (since 0 was in *partials*[3] in the previous step), returning 6 and 12 as answers to $Q1$ and $Q2$ respectively.

Skipping further, in Step 7 *SlickDeque* (Inv) adds 2 to both answers, and subtracts 1 from $Q1$'s answer (since it is now out of range for $Q1$) making it 9, and subtracts 5 from $Q2$'s answer (since 5 was in *partials*[1] in the previous step) making it 10. ■

Notice that in this example *Naive* had to execute a total of 48 Sum operations, while *SlickDeque* (Inv) executed a total of 32 operations (Sum and Subtract).

SlickDeque for Non-Invertible Aggregates

For processing non-invertible aggregates we propose a novel algorithm, *SlickDeque* (Non-Inv), which accelerates the processing of *ACQs* by intelligently maintaining and utilizing a deque data structure consisting of nodes allocated in chunks interconnected with pointers. For simplicity of explanation we assume

that each node is allocated on a separate chunk. The benefits of allocating multiple nodes per chunk are explained in Section 4.2. Pseudocode for *SlickDeque* (Non-Inv) is depicted in Algorithm 2, and similarly to *SlickDeque* (Inv) it consists of two major phases: *Preparation* and *Execution*.

The Preparation Phase Similarly to *SlickDeque* (Inv), the execution starts by building a *sharedPlan* by executing the function *buildSharedPlan* (line 4). It is constructed using one of the partial aggregation techniques as discussed in Section 2.1, and it includes a full list of partials augmented with their lengths and lists of queries that need to be evaluated for each partial. The query with the longest range in terms of the number of partials is identified and saved as the member *wSize* of the *sharedPlan*, signifying the necessary window length needed to process all input queries.

After generating the *sharedPlan*, *SlickDeque* (Non-Inv) defines node, *Node*, structure that has members *pos* and *val*, and initializes deque, *d*, composed of nodes, *Node*, (lines 6-7). *SlickDeque* utilizes the *currPos* variable to signify the sequential number of the current partial aggregate. It starts at 0 initially and increases to *wSize* - 1 during execution, after which it wraps back to 0.

The Execution Phase is implemented as a loop that continuously returns all query results while they are expected, and identically to *SlickDeque* (Inv), it begins by aggregating a *newPartial*. The if-statement on line 13 is removing the expired node (if present) from the head of the deque, *d*. The while-loop after that (line 16) is executing operation \oplus on two values: the value of the tail node and of the new partial. If the new partial is returned by the operation, the tail node is removed from the deque (it will never be a query answer), and the next one is tested, otherwise the loop stops. The new node is then added to the deque with *currPos* as the position and *newPartial* as the value (line 19).

Next, set *queriesToAnswer* (a subset of Q scheduled at this position) is accessed from the *sharedPlan*, and the answers for its queries are produced in the for-loop below. Naturally, when the *sharedPlan* was constructed, all queries in each *queriesToAnswer* set were ordered descendingly by their range. We utilize this ordering to answer all queries by looping over the deque only once, since the larger ranges always correspond to the deque nodes closest to the head. Therefore, the position *i* within the deque is defined outside the loop and initialized to the head of the deque (line 21).

The loop starts by identifying the *startPos* of the aggregation for each query, q , by subtracting q 's range from *currPos* (line 23). If *startPos* is negative it means that this range crosses a boundary between two windows, and thus the boolean *boundaryCrossed* is set to true and *startPos* is increased by the *wSize*. Otherwise *boundaryCrossed* is set to false.

Then, based on whether the current range crosses the window boundary or not, one of the two subsequent *Answer Loops* is executed (lines 29-39), iterating over nodes from the current position *i* until the answer node is identified based on the *pos* member of each node, and returned as an answer to the query, q . The next iteration (to answer the next query) will continue working from the position *i* forward, until all queries are processed. After returning all required answers the *currPos* is moved one position forward (lines 42-45).

The following Example 3 (illustrated in Fig. 9) should clarify the above algorithm. To make the explanation more intuitive we again execute the two queries $Q1$ and $Q2$ on the same incoming datastream using *Naive* and *SlickDeque* (Non-Inv), and illustrate each step of their processing side-by-side.

Algorithm 2 SlickDeque (Non-Inv) Pseudocode

```

1: Input: A set of aggregate continuous queries  $Q$ , non-invertible aggregate operation  $\oplus$ , and partial aggregation technique PAT
2: Output: Continuous answers to queries in  $Q$  according to their specifications.
3:           Phase 1 (Preparation)
4: sharedPlan = buildSharedPlan( $Q$ , PAT)
5: wSize = sharedPlan.wSize
6: Node with members pos and val
7: Deque d composed of nodes of type Node
8: currPos = 0
9:           Phase 2 (Execution)
10: while results are expected do
11:   length = sharedPlan.getNextPartialsLength()
12:   newPartial = partialAggregator.aggregate(length, PAT)
13:   if d.size > 0 AND d.front.pos == currPos then
14:     d.pop_front()
15:   end if
16:   while d.size > 0 AND d.back.val  $\oplus$  newPartial == newPartial do
17:     d.pop_back()
18:   end while
19:   d.push_back(new Node(currPos, newPartial))
20:   queriesToAnswer = sharedPlan.getNextSetOfQueries()
21:   i = d.firstNode
22:   for each query q in queriesToAnswer do
23:     startPos = currPos - q.range
24:     boundaryCrossed = false
25:     if startPos < 0 then
26:       startPos += wSize
27:       boundaryCrossed = true
28:     end if
29:     if boundaryCrossed == false then
30:       //Answer Loop 1
31:       while i.pos < startPos OR i.pos > currPos do
32:         i = i.nextNode
33:       end while
34:     else
35:       //Answer Loop 2
36:       while i.pos < startPos AND i.pos > currPos do
37:         i = i.nextNode
38:       end while
39:     end if
40:     send i.val as answer to q
41:   end for
42:   currPos++
43:   if currPos == wSize then
44:     currPos = 0
45:   end if
46: end while

```

Example 3 Assume we have queries $Q1$ and $Q2$, which are seeking Max over the ranges of 3 and 5 tuples respectively, both with a slide of 1 tuple. The slide size is again set to one tuple for simplicity, which means that there is no partial aggregation and the answers to both queries need to be calculated after every new tuple arrival. As before, the range of $Q2$ (5) is greater than the range of $Q1$ (3), and the slides of $Q1$ and $Q2$ are the same, the shared execution plan has a $wSize$ of 5 tuples.

While *Naive* uses the circular *partials* array to maintain the incoming partials (in this case just tuples), *SlickDeque* (Non-Inv) only utilizes deque in its operation. In both *partials* and deque we mark the positions modified in each step. The tuples enter the system in the same order as in Example 2: 6, 5, 0, 1, 3, 4, 2, 7.

After the initialization Step, in Step 1 the first tuple, 6, arrives. *Naive* stores it at the $currPos$ in the *partials* array, and iterates

Step	Naive	SlickDeque (Non-Inv)	Answer
0	partials [0 1 2 3 4] [-∞ -∞ -∞ -∞ -∞]	deque []	Q1 Q2 n/a n/a
1	partials [0 1 2 3 4] [6 -∞ -∞ -∞ -∞]	deque [0] [6]	6 6
2	partials [0 1 2 3 4] [6 5 -∞ -∞ -∞]	deque [0 1] [6 5]	6 6
3	partials [0 1 2 3 4] [6 5 0 -∞ -∞]	deque [0 1 2] [6 5 0]	6 6
4	partials [0 1 2 3 4] [6 5 0 1 -∞]	deque [0 1 3] [6 5 1]	5 6
5	partials [0 1 2 3 4] [6 5 0 1 3]	deque [0 1 4] [6 5 3]	3 6
6	partials [0 1 2 3 4] [4 5 0 1 3]	deque [1 0] [5 4]	4 5
7	partials [0 1 2 3 4] [4 2 0 1 3]	deque [0 1] [4 2]	4 4
8	partials [0 1 2 3 4] [4 2 7 1 3]	deque [2] [7]	7 7

Figure 9: Example 3 processing of non-invertible aggregate queries $Q1$ and $Q2$ using Naive and SlickDeque algorithms.

over the last 3 indexes (3, 4, and 0) to answer $Q1$, and over the entire array to answer $Q2$. Both answers in this case are 6.

SlickDeque (Non-Inv) places a new node with $pos = 0$ (which is $currPos$) and $val = 6$, at the head of the deque, and since its pos value is both within the last 3 and 5 positions from $currPos$, its val is returned as the answer to both $Q1$ and $Q2$.

In Step 2, the new partial, 5, is placed into the $currPos$, and *Naive* iterates again over the past 3 tuples to answer $Q1$ and over the whole window to answer $Q2$, and returns the Max value from all values visited, which is 6. Our algorithm on the other hand, places the new tuple 5 as a val of the new node (with $pos = 1$) at the end of the deque, and returns 6 (the val of the head node of the deque) as an answer to both queries.

Skipping ahead, in Step 4 *SlickDeque* (Non-Inv) removes the tail node of the deque since the newly arrived tuple, 1, is greater than 0, which is the val of the tail node, and adds the new node with $pos = 3$ and $val = 1$ at the end of the deque. Since $Q2$ has a larger range, it is scheduled to be processed first. Its $startPos$ is identified: $3 - 5 = -2$, and since -2 is negative, the window boundary is crossed. Therefore $startPos$ is moved to $-2 + 5 = 3$, and the *Answer Loop 2* is executed returning the val of the head node, 6. The $startPos$ of $Q1$ is $3 - 3 = 0$, and since 0 is not negative, the window boundary is not crossed. Thus, the answer is produced by iterating using *Answer Loop 1*, which returned 5, the val of the second node from the head.

Skipping further, in Step 6 *SlickDeque* (Non-Inv) removes the head node of the deque (with $pos = 0$ and $val = 6$) which expires at this step since the $currPos$ is 0. Also, since the newly arrived tuple, 4, is greater than 3, the last node of the deque is removed, and the new node with $pos = 0$ and $val = 4$ is added at the end

of the deque. Q_2 and Q_1 are then both processed by executing the *Answer Loop 2* and returning 5 and 4 respectively. ■

Note that this example also shows the advantage of *SlickDeque* (Non-Inv) over *Naive* by showing that *Naive* had to execute 48 Max operations total, while *SlickDeque* (Non-Inv) executed 11.

4 COMPLEXITY ANALYSIS

In this section, we calculate the time and space complexities of *Naive*, *B-Int*, *FlatFAT*, *FlatFIT*, *TwoStacks*, *DABA*, and *SlickDeque*. These are summarized in Table 1.

4.1 Time Complexities

We evaluate each algorithm’s time complexity in terms of the number of aggregate operations it performs per slide to return all query answers given a window size of n partial aggregates. This metric was chosen because the aggregate operations are (1) applied directly to the input data, (2) constitute the the bulk of all performed operations, and (3) their number correlates best with the actual query performance. In order to cover the entire complexity space, we calculate **amortized** complexities as well as **worst-case** complexities. Amortized complexities are important to us because they correlate with *ACQ* processing throughputs, while worst-case ones reflect possible latency spikes.

In addition to providing calculations for a **single query** environment (where only one query covering the entire window is executed each slide), we also evaluate a multi-query environment with the maximum number of queries (which we refer to as a **max-multi-query** environment). This way, a single query environment can be thought of as a **lower bound** of complexity per slide, while a max-multi-query environment (which executes all queries covering all possible ranges from 1 to the window length (n) each slide), can be thought of as the **upper bound**. It is clear that in most cases the complexity of the general case (with any other numbers of queries) lays between these bounds.

Naive has an exact time complexity (with matching amortized and worst cases) because it always executes the same number of operations per slide. In a single query environment, its complexity is $n - 1$ (asymptotically n) because it simply iterates over all n partials and aggregates them.

In a max-multi-query environment, *Naive* needs to return n answers each slide for ranges from 1 to n , yielding 0 to $n - 1$ operations, respectively. By summing up this arithmetic sequence we get $\frac{n^2}{2} - \frac{n}{2}$ (asymptotically n^2).

FlatFAT has an exact time complexity of $\log_2(n)$ in a single query environment since each new partial updates the binary tree in a bottom-up fashion from the leaf to the root. Since the number

of levels in a binary tree is $\log_2(n) + 1$, *FlatFAT* needs exactly $\log_2(n)$ operations to calculate the query answer. In a max-multi-query environment it is intuitive that the upper bound of the time complexity is $n \cdot \log_2(n)$, since *FlatFAT* needs to iterate over n different query ranges at each slide and each range would require $\log_2(n)$ operations at most to return the result. The exact complexity per slide can be produced by iterating over all possible ranges and summing their required numbers of operations, which equates to: $n \cdot \log_2(n) - \frac{3n}{2} + \frac{5\log_2(n)}{2} + \frac{5}{2}$. For simplicity, we use the asymptotic equivalent of this complexity: $n \cdot \log(n)$.

B-Int similarly to *FlatFAT* is of a binary nature, and is only different in how it handles updates and look-ups. In [29] *B-Int* has been shown to have the same asymptotic time complexity as *FlatFAT*, with *B-Int* being slower by a constant factor, which we confirm in this work as well.

FlatFIT executes different numbers of operations for different slides, unlike *Naive*, *FlatFAT*, and *B-Int*, which causes spikes in latency. The execution of *FlatFIT* follows a cyclical pattern which repeats every $n + 1$ slides, where n is the window size. In a single query environment, the so called *window reset* event happens once per such period and constitutes the worst-case complexity per slide. During the *window reset* the indexes of the entire data structure are updated in $n - 1$ steps. The *window reset* operation is surrounded by two slides that require just one operation, and the rest of the slides in a period require two operations each. By summing everything, we have the amortized complexity for the natural period of *FlatFIT*: $(n - 1) + 2(n - 2) + 2 = 3(n - 1)$, equating to $3n$ operations for the period of n slides, which in turn makes the amortized complexity asymptotically constant and equal to 3 operations per slide.

In a max-multi-query environment, *FlatFIT* keeps the data structure maximally updated by answering queries over all possible ranges each slide, which allows it to calculate the query answers with just one or zero operations each. Due to this, the *window reset* event happens only once at the beginning of the execution phase, and therefore in this scenario the operational complexity of the *FlatFIT* algorithm is not amortized and yields $n - 1$ operations per slide (asymptotically n).

TwoStacks also executes different numbers of operations for different slides, which introduces latency spikes similarly to *FlatFIT*. During insertions, each new partial is added to the *B* stack and one aggregate operation is performed to determine the new aggregate value of the entire stack *B*. After that, another operation is performed using the top values of both the *F* and *B* stacks to return the query answer, which makes the complexity of insertions 2 operations. The majority of evictions are free since they are done by just popping the node from the *F* stack. When *F* becomes empty, however, *B* is flipped onto *F* by popping values one-by-one from *B* and inserting them into *F* while performing one aggregate operation per insertion (to populate *agg* values on *F*). The flip procedure (n operations) clearly constitutes the worst-case complexity per slide. To calculate the amortized complexity we add all operations per one full iteration of the algorithm: n insertions (1 operations each), n queries (1 operation each), and one eviction that causes stack flip procedure (n operations), totalling $3n$ operations per n slides. Thus, the amortized complexity of the algorithm in constant and equals 3 operations per slide. *TwoStacks* does not currently allow multi query processing.

DABA was proposed to alleviate latency spikes in *TwoStacks* by making its worst-case time complexity constant (though it still performs different numbers of operations each slide). By doing

Table 1: Algorithmic Complexities

Algorithm	Time			Space	
	Single Query	Max-Multi Query	Max-Multi Query	Single Query	Max-Multi Query
	Amort			Worst	Single Query
Naive	n	n	n^2	n	n
FlatFAT	$\log(n)$	$\log(n)$	$n \cdot \log(n)$	$2n^{**}$	$2n^{**}$
B-Int	$\log(n)$	$\log(n)$	$n \cdot \log(n)$	$2n^{**}$	$2n^{**}$
FlatFIT	3	n	n	$2n$	$2n$
TwoStacks	3	n	—	$2n$	—
DABA	5	8	—	$2n$	—
Slick Deque	Inv	2	2	$2n$	n
	Non-Inv	<2	n^*	n	2 to $2n^*$

*the probability of these cases is negligible: 1 in $n!$.

**true only when n is a power of 2, otherwise $3n$.

that *DABA* sacrifices its amortized time complexity (and consequently its throughput). Per one full window iteration *DABA* executes 2 flip actions, n shift actions, and n evict actions (which all cost 0 operations), n shrink actions (costing 3 operations each), and also n insert actions and n answer look-up actions (cost 1 operation apiece), totalling $5n$ operations per n slides, which yields the amortized complexity of 5 operations. *DABA*'s worst-case complexity can be attributed to a step that performs the following sequence of actions: Evict, Flip, Shrink, Insert, Shrink, Query, which costs 8 operations total. Similarly to *TwoStacks* *DABA* does not currently support multi query processing.

SlickDeque for Invertible Operations has an exact time complexity of just 2 operations per slide in a single query environment, since after each arrival of the new partial aggregate, the query answer is updated twice: once by executing an aggregate operation with the incoming partial, and once by executing the inverse operation with the expiring partial. In a max-multi-query environment *SlickDeque* (Inv) has to perform $2n$ operations, since one aggregate operation and one inverse operation need to be executed on each of the answers to n queries, which makes the algorithm's exact time complexity $2n$.

SlickDeque for Non-Invertible Operations executes variable numbers of operations per slide. As opposed to *FlatFIT*, *TwoStacks*, and *DABA* which are input agnostic and have their worst-case steps executed periodically, *SlickDeque* (Non-Inv) depends on the input, and the probability of ever executing its worst-case step is minuscule as we point out below.

Intuitively, in the long-running environment with a non-infinite window, each partial can cause at most two operations: one when it is inserted (invokes its comparison with the tail of the deque), and one when it is deleted by another incoming partial (invokes comparison of the incoming partial with the next item on deque). Clearly, the only two situations when a partial performs less than two operations in its lifetime are (1) if it becomes the first element of the deque after its insertion (either by removing all other partials or by being inserted into an empty deque), or (2) if it expires before being removed by another partial. If both situations happen to the same partial it will be involved in 0 operations in its lifetime. Also, it is impossible to execute a full window iteration without hitting one of the two situations by one of the partials at least once, since we cannot have an element in a deque that would both not get removed by another incoming partial as well as not expired after a full window iteration. Thus, the amortized complexity of this algorithm depends on the input, however it is always less than 2 operations.

The worst time complexity of this algorithm happens when the input (except the last partial of the window) is ordered in the opposite way of the aggregate operator order, e.g., if Max is processed and the entire input is ordered descendingly, forcing the deque to fill up, after which the next input partial has the largest value so far. This causes the new element to perform n operations while deleting all nodes on the deque. Fortunately, such a situation is highly unlikely on most inputs (1 in $n!$ chance in the uniform case). Consider the state-of-the-art *DABA* algorithm that we showed to have a worst-case complexity of 8 operations. In order for *SlickDeque* (Non-Inv) to have a step with the same complexity there should be at least 9 ordered partials in the input. The probability of receiving 9 values ordered in a specific way in a row is 1 out of $9!$ (equals 362880), which is highly unlikely.

In a max-multi-query environment, to process all queries scheduled at a slide, the deque is traversed from the head while

answering each query. Clearly, if the number of nodes in the deque is smaller than the number of different queries to answer, some nodes will have answers to multiple queries. Thus, the worst case would again be when the input forced the deque to completely fill up, for which the probability is again 1 in $n!$. In such a case, iterating over the entire deque at each step will take n operations (and at worst 2 operations per step as shown in the single query environment), so the complexity of the worst-case becomes $2n$. In the best case, the deque would have only one node each slide that would answer all queries, which would make complexity just 2 operations total.

Summary The differentiated processing of invertible and non-invertible operations allows *SlickDeque* to utilize optimizations tailored towards each type that are not available in the general case. Thus, *SlickDeque* is superior in the time complexity for both invertible and non-invertible cases compared to all other algorithms. However, in the worst-case complexity per slide, theoretically *SlickDeque* has a small possibility (1 in 362880 based on the input) to be outperformed by *DABA*.

4.2 Space Complexities

Naive has the space complexity of n since it stores partials only once and does not keep any additional structures. This complexity stands despite the number of registered queries, since additional queries do not require any additional structures.

FlatFAT and **B-Int** both have the space complexity of $2^{\lceil \log(n) \rceil + 1}$. Due to their binary nature, they are more space efficient when the window size is a power of two, in which case they consume $2n$ of memory: n for all leaf nodes and $n - 1$ for all tree nodes above leaves. The first position within a flat array is normally left unused in order to simplify the addressing of nodes within the tree. In cases where the window size is not a power of two, *FlatFAT* and *B-Int* round it up to the closest power of two, which is mathematically expressed as: $2^{\lceil \log(n) \rceil}$. Therefore, the space complexity of these algorithms yields $2^{\lceil \log(n) \rceil + 1}$. The window rounding manifests the worst-case space complexity of $3n$.

FlatFIT needs two pre-allocated arrays of size n to operate and a stack that can grow up to 2 values total in a single query environment and in a max-multi-query environment. This results in an asymptotic space complexity of *FlatFIT* $2n$. However, in terms of space complexity, single query and max-multi-query environments do not bound *FlatFIT*. In a general case where we have more than one query and less than the maximum queries registered, the stack might have to store up to $n/2$ values (case with two queries) at most. However, each additional query (of a different range) after that cuts the maximum stack memory consumption in half. Therefore, if the number of queries is q , the space complexity of *FlatFIT* becomes $2n$ for $q = 1$ and $q = n$, and $2n + \frac{n}{2^{q-1}}$ for the rest of the possible values of q .

TwoStacks uses stack structures with nodes containing two values, however both stacks combined can never have more than n nodes total by the nature of the algorithm, which makes its space complexity $2n$.

DABA similarly to *TwoStacks* maintains the front and back stacks with nodes consisting of both values and aggregates, however it is implemented on top of the doubly linked list of chunks. The space complexity of *DABA* depends on the number of underlying chunks, specifically, having less chunks that are bigger in size saves space on pointers (left and right), but wastes space on over-allocations (periodically window slides between chunks during

the execution leaving up to two chunks' worth of space wasted). If the window is split into k chunks, then *DABA*'s space complexity is: $2n + 4k + 4n/k$. If we take a derivative with respect to k , equate it to zero, and solve for k , we conclude that the minimum space complexity for *DABA* is achieved by setting k to \sqrt{n} , and it equals $2n + 4\sqrt{n}$ (asymptotically $2n$).

SlickDeque for invertible operations stores partial aggregates similarly to *Naive*. In addition, it stores the answer for each query with a unique range, making its single query space complexity $n + 1$, and max-multi-query $2n$.

SlickDeque for non-invertible operations performs node allocations in chunks to reduce the space required by pointers similarly to *DABA*, causing an overallocation of up to two chunks' worth of space (at the beginning and at the end of the deque). The space complexity of *SlickDeque* (Non-Inv) does not depend on the number of registered queries, but depends on the input. In the worst-case, the input forces the deque to become full. In such a case, having n nodes with two values each, and k chunks with two pointers each, the space consumption becomes $2n + 4k + 4n/k$. By taking a derivative with respect to k , equating it to zero, and solving for k , we conclude that k should be set to \sqrt{n} to minimize the worst-case complexity, which becomes $2n + 4\sqrt{n}$ (asymptotically $2n$). Similarly to the time complexity, the chance of the worst-case happening in normal conditions is very low: just 1 in $n!$. In the best case, however, each incoming partial forces the deque to eliminate all of its nodes, making the space complexity constant (2).

Summary *SlickDeque* shows a clear advantage over the rest of the algorithms in terms of space complexity. *SlickDeque* (Inv) shares the space complexity of n with *Naive*, while the rest of the algorithms have a complexity of at least $2n$, and the complexity of *SlickDeque* (Non-Inv) is always less or equal than $2n$ (based on the input). This means that only *Naive* can possibly outperform it, however the probability of that happening is low (just 1 in $n!/2$), and even then, *Naive* is still not a feasible solution because of its high time complexity.

5 EXPERIMENTAL EVALUATION

In this section, we present our experimental evaluation that confirms the theoretical superiority of *SlickDeque* in practice, by comparing it to other final aggregation approaches.

5.1 Experimental Testbed

Platform In order to test the performance of our sliding-window aggregation technique, we built an experimental platform in C++ (compiled with G++5.4.1). Specifically, we implemented a stand-alone stream aggregator platform and programmed the *Naive*, *FlatFAT*, *B-Int*, *FlatFIT*, *TwoStacks*, *DABA*, and *SlickDeque* (Inv and Non-inv) algorithms within the same codebase, sharing data structures and function calls to enable a fair comparison. Although all of the compared algorithms can be easily ported to any commercial general purpose stream processing system, we chose to go with a stand-alone platform to carry out our evaluation in an isolated environment in order to avoid any potential system interference and overheads. In the future we are planning to repeat our evaluation on a production system.

Dataset We utilized the DEBS12 Grand Challenge Dataset [16] which contains events generated by sensors of large hi-tech manufacturing equipment. Each tuple in this dataset incorporates 3 energy readings and 51 values signifying various sensor states. The records were sampled at the rate of 100Hz, and the whole

dataset includes ~33 million unique events, which we made into a dataset of 134 million tuples.

Workload Clearly, the performance of the final aggregation techniques heavily depends on the window size, i.e., the larger the window size the longer it takes to process updates to it. Thus, we varied the window size from 1 tuple to 134 million tuples, which is the maximum window size with our dataset. Given that the goal of our evaluation is just to compare different final aggregation techniques, we eliminated any side effects (i.e., overheads or benefits) induced by partial aggregation by setting all query slides to one tuple.

Evaluation Metrics We chose to compare the algorithms using throughput, latency, and memory requirement. *Throughput* is measured as the number of query results returned per second in a single query environment, while in a multi-query environment it is measured as the number of slides of a shared execution plan processed per second. *Latency* is measured in terms of the total time it took to calculate and return the answer to each query. *Memory Requirement* is measured by the maximum resident set size of processes running the corresponding techniques.

5.2 Experimental Results

We ran our experiments on an Intel(R) i7-4770 CPU @ 3.40GHz with 16 GB of RAM. For robustness, all the results were averaged over three independent runs of each experiment aggregating three different energy readings from the DEBS12 dataset.

Exp 1: Single Query Throughput

Exp1(a) Invertible Aggregates (Fig. 10)

In this experiment we varied the window size from 1 to 134 million tuples where each window is a power of two, and ran a query calculating the invertible aggregation Sum over the entire window after each new tuple arrival. From the results in Fig. 10 we clearly see that there are two groups of algorithms based on their behavior with increasing window size: (1) with constant throughput (*SlickDeque*, *FlatFIT*, *TwoStacks*, and *DABA*), and (2) with steadily degrading throughput (*FlatFAT*, *B-Int*, and *Naive*). Notice that the throughput rates are similar to what we expected from the theoretical analysis of the algorithms in Section 4.

Fig. 10 shows that *SlickDeque*'s throughput is on average 15% higher than the throughput of the second best algorithm (*FlatFAT* on windows 1 through 16, and *FlatFIT* on the rest) with a maximum of 19%. We also observed that *SlickDeque* starts outperforming other algorithms on windows as small as 4 tuples and increases its gain rapidly. *FlatFAT* showed to be more beneficial than *SlickDeque* only on window sizes from 1 to 4 tuples, however this benefit is negligible (1% at max).

Exp1(b) Non-Invertible Aggregates (Fig. 11)

In this experiment we replaced the calculation of Sum with the non-invertible aggregation Max, that again runs over the entire window after each tuple arrival. Similarly to Exp1(a), we see that the throughput of some algorithms is practically unaffected by the increasing window size. The results are depicted in Fig. 11. Once again, the throughput rates correspond to what we expected from the theoretical analysis of the algorithms.

In this experiment *SlickDeque*'s throughput is on average 7% higher than the throughput of the second best algorithm with a maximum of 10%, and *SlickDeque* starts outperforming all other algorithms on windows as small as 16 tuples. *FlatFAT* showed to be more beneficial than *SlickDeque* only on window sizes from 1 to 8 tuples with an advantage of 7% at max.

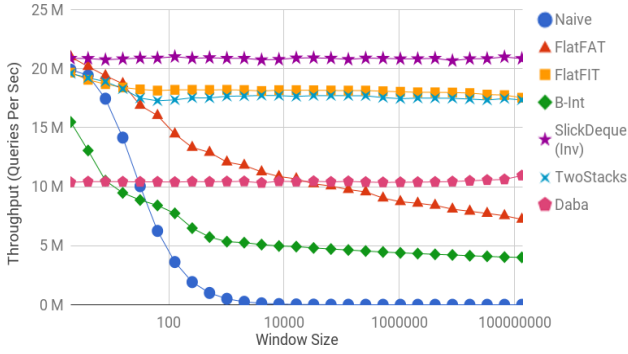


Figure 10: Exp 1 Throughput in processed queries per second in single query environment (Sum)

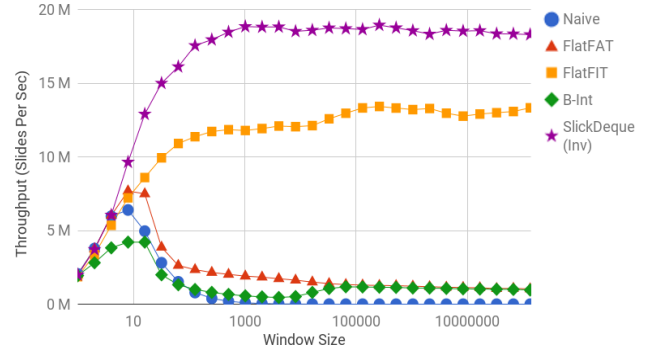


Figure 12: Exp 3 Throughput in processed slides per second in multi-query environment (Sum)

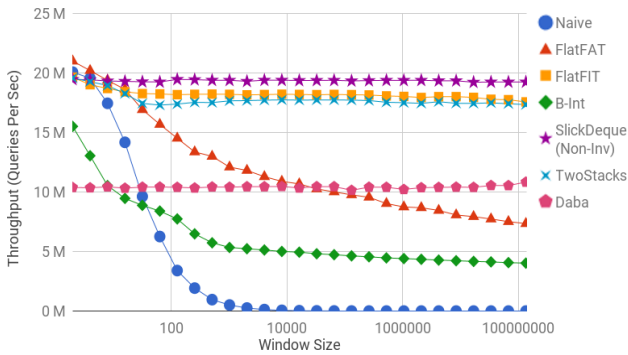


Figure 11: Exp 2 Throughput in processed queries per second in single query environment (Max)

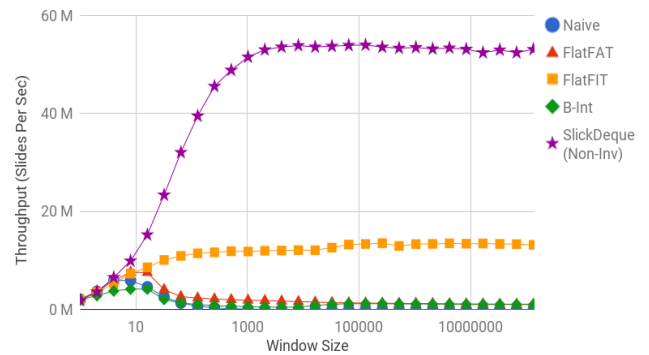


Figure 13: Exp 4 Throughput in processed slides per second in multi-query environment (Max)

Exp 2: Max-Multi-Query Throughput

Exp2(a) Invertible Aggregates (Fig. 12)

In this experiment we ran a maximum number of queries calculating Sum value over the ranges from 1 to the window size after each new tuple arrives. In this context increasing the window also increases the number of queries that are processed after each slide, enabling higher reuse of unchanged partial results among them. Thus, in Fig. 12 we see that the throughput gradually increases until the moment when the overhead of dealing with the large window outweighs the benefit of sharing between queries.

In this setting, our approach demonstrated superior scalability yet again by yielding throughput that is on average 45% higher than the throughput of the second best technique with a maximum of 60%. Notice that *SlickDeque* performs the best on window sizes from 4 tuples to 134 million tuples and only underperforms compared to other algorithms on window sizes 1 and 2 by 3% and 2%, respectively.

Exp2(b) Non-Invertible Aggregates (Fig. 13)

In this experiment we ran the maximum number of queries calculating Max over all ranges from 1 to the entire window after each tuple arrival. The results are depicted in Fig. 12, and are close to our results in experiment Exp2(a).

In this setting *SlickDeque* yielded throughput on average 266% higher than the throughput of the second best technique with a maximum of 345%. *SlickDeque* showed to perform the best on windows from 4 tuples to 134 million tuples while falling behind *Naive* and *FlatFAT* on windows 1 and 2 by 7% on average.

Summary In all throughput experiments *SlickDeque* exhibits the best results, while being slightly outperformed on small window

sizes (between 1 and 8 tuples) when the overhead of maintaining its structure outweighed the benefit of using it.

Exp 3: Query Processing Latency (Fig. 14)

In this experiment we fixed our window size at 1024 tuples and ran all algorithms on the first million tuples of the DEBS data set while recording how long it took to return an answer to each query. We executed a single query processing Sum (invertible) in the first test, and Max (non-invertible) in the second test. We dropped the highest 0.005% latencies from all algorithms as outliers. The latency results of both tests were nearly identical for all algorithms except *SlickDeque*, thus we combined them in Fig. 14, where only *SlickDeque* has separate entries for invertible and non-invertible cases.

Fig. 14 shows that both invertible and non-invertible *SlickDeque* versions exhibited the lowest latency in all the following categories: Min, Max, Average, Median, 25th Percentile, and 75th Percentile. Across all of the abovementioned categories, *SlickDeque* outperformed the second best algorithm by 8% on average and 17% at most (for the non-invertible version), and by 75% average and 548% at most (for the invertible version). Also, *SlickDeque* outperformed the second best *DABA* algorithm by 283% on average in terms of the lowest max latency spike.

Exp 4: Memory Requirement (Fig. 15)

In this experiment we again varied the window size from 1 tuple to 134 million tuples (but also included window sizes that are not powers of two). We executed a query calculating the invertible Sum aggregation in the first experiment, and the non-invertible Max aggregation in the second. We measured the maximum

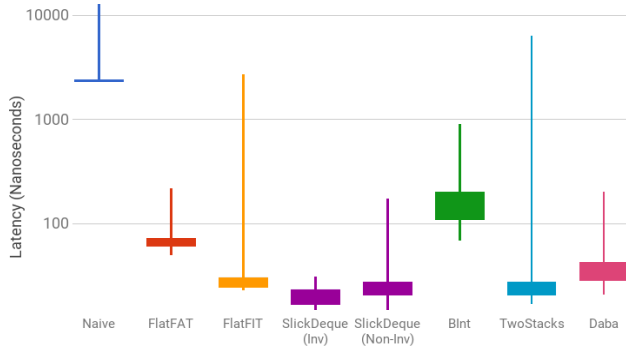


Figure 14: Exp 4 Latency in nanoseconds per query answer

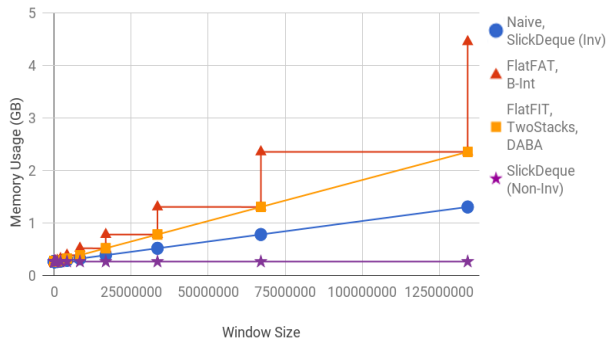


Figure 15: Exp 5 Experimental Memory Usage in Gigabyte increments

resident set size (RSS) of the processes for all runs. The results of this test are depicted in Fig. 15. On this graph, we combined the results of both invertible and non-invertible runs of all algorithms since their space requirements were identical in both Sum and Max cases except for *SlickDeque*, which we plotted separately for each case. Notice that due to the great similarity of space requirement for several algorithms, we plotted: *FlatFAT* together with *B-Int*, *FlatFIT* together with *TwoStacks* and *DABA*, *Naive* together with *SlickDeque (Inv)*, and *SlickDeque (Non-Inv)* was plotted separately. The memory requirement rates correspond to what we predicted from the theoretical analysis in Section 4. *SlickDeque* demonstrated excellent scalability by matching the space usage of *Naive* for the invertible case, and for the non-invertible one outperforming the second best algorithm (*Naive*) by 2 times on average with a maximum of 5 times.

6 CONCLUSIONS

The key contribution of this paper is *SlickDeque*, a novel technique for incremental sliding-window final aggregation processing for single- and multi-query environments. Its power is the differentiated handling of aggregate operations based on their invertibility, which allows *SlickDeque* to use optimizations tailored towards each type and that are not available in the general case.

We theoretically showed that *SlickDeque* significantly decreases the number of operations required for a continuous query to return results while reducing its space requirement. As far as we know, there are no prior algorithms that can achieve the same time and space complexities without loss of query generality. We showed experimentally that *SlickDeque* achieves up to 3.5x higher throughput compared to the state-of-the-art algorithms,

while maintaining up to 5.5x lower latency and utilizing up to 5x less memory. Our next step is to evaluate *SlickDeque* in dynamic and multi-node environments on production systems.

Acknowledgments Research reported in this publication was partially supported by the National Institutes of Health under Award U01HL137159 and gift from EMC. The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Institutes of Health or EMC.

REFERENCES

- [1] D. J. Abadi et al. Aurora: a new model and architecture for data stream management. *VLDBJ*, 2003.
- [2] D. J. Abadi et al. The design of the borealis stream processing engine. In *CIDR*, 2005.
- [3] T. Akidau et al. Millwheel: Fault-tolerant stream processing at internet scale. In *VLDB*, 2013.
- [4] A. Arasu and G. S. Manku. Approximate counts and quantiles over sliding windows. In *SIGMOD*, 2004.
- [5] A. Arasu and J. Widom. Resource sharing in continuous sliding-window aggregates. In *VLDB*, 2004.
- [6] S. Badiozamani, K. Orsborn, and T. Risch. Framework for real-time clustering over sliding windows. In *SSDBM*, 2016.
- [7] A. Bulut and A. K. Singh. Swat: Hierarchical stream summarization in large networks. In *DataEngConf*, 2003.
- [8] P. Carbone, J. Traub, A. Katsifodimos, S. Haridi, and V. Markl. Cutty: Aggregate sharing for user-defined windows. In *CIKM*, 2016.
- [9] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. *SIAM journal on computing*, 2002.
- [10] T. M. Ghanem, M. A. Hammad, M. F. Mokbel, W. G. Aref, and A. K. Elmagarmid. Incremental evaluation of sliding-window queries over data streams. *TKDE*, 2007.
- [11] P. B. Gibbons and S. Tirthapura. Distributed streams algorithms for sliding windows. In *SPAA*, 2002.
- [12] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data mining and knowledge discovery*, 1997.
- [13] S. Guirguis, M. Sharaf, P. K. Chrysanthis, and A. Labrinidis. Three-level processing of multiple aggregate continuous queries. In *ICDE*, 2012.
- [14] S. Guirguis, M. A. Sharaf, P. K. Chrysanthis, and A. Labrinidis. Optimized processing of multiple aggregate continuous queries. In *CIKM*, 2011.
- [15] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez. Streamcloud: An elastic and scalable data streaming system. *TPDS*, 2012.
- [16] Z. Jerzak, T. Heinze, M. Fehr, D. Gröber, R. Hartung, and N. Stojanovic. The debs 2012 grand challenge. In *DEBS*, 2012.
- [17] M. Kaufmann, A. A. Manjili, P. Vagenas, P. M. Fischer, D. Kossmann, F. Färber, and N. May. Timeline index: A unified data structure for processing queries on temporal data in sap hana. In *SIGMOD*, 2013.
- [18] S. Krishnamurthy, C. Wu, and M. Franklin. On-the-fly sharing for streamed aggregation. In *SIGMOD*, 2006.
- [19] J. Li, D. Maier, K. Tuft, V. Papadimos, and P. A. Tucker. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *SIGMOD*, 2005.
- [20] J. Li, D. Maier, K. Tuft, V. Papadimos, and P. A. Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *SIGMOD*, 2005.
- [21] B. Moon, I. F. V. López, and V. Immanuel. Scalable algorithms for large temporal aggregation. In *DataEngConf*, 2000.
- [22] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, R. Varma. Query processing, approximation, and resource management in a data stream management system. In *CIDR*, 2003.
- [23] D. Piatov and S. Helmer. Sweeping-based temporal aggregation. In *SSTD*, 2017.
- [24] A. U. Shein, P. K. Chrysanthis, and A. Labrinidis. F1: Accelerating the optimization of aggregate continuous queries. In *CIKM*, 2015.
- [25] A. U. Shein, P. K. Chrysanthis, and A. Labrinidis. Processing of aggregate continuous queries in a distributed environment. In *BIRTE*, 2015.
- [26] A. U. Shein, P. K. Chrysanthis, A. Labrinidis. Flatfit: Accelerated incremental sliding-window aggregation for real-time analytics. In *SSDBM*, 2017.
- [27] E. Soisalon-Soininen and P. Widmayer. Single and bulk updates in stratified trees: An amortized and worst-case analysis. In *Computer Science in Perspective*. Springer, 2003.
- [28] K. Tangwongsan, M. Hirzel, and S. Schneider. Low-latency sliding-window aggregation in worst-case constant time. In *DEBS*, 2017.
- [29] K. Tangwongsan, M. Hirzel, S. Schneider, and K.-L. Wu. General incremental sliding-window aggregation. *VLDB*, 2015.
- [30] A. Toshniwal et al. Storm@twitter. In *SIGMOD*, 2014.
- [31] J. Yang and J. Widom. Incremental computation and maintenance of temporal aggregates. In *DataEngConf*, 2001.

Modeling and Exploiting Goal and Action Associations for Recommendations

Dimitra Papadimitriou
University Of Trento
Trento, Italy
papadimitriou@disi.unitn.it

Yannis Velegrakis
University Of Trento
Trento, Italy
velgias@disi.unitn.eu

Georgia Koutrika
Athena Research Center
Athens, Greece
georgia@imis.athena-innovation.gr

ABSTRACT

Recommender systems are used to identify those items in a large collection that are more likely to be of interest to a user. A common principle of most recommenders is that whatever happened in the past is a good indicator of the future. We offer a different perspective. Considering the fact that in real life users do their selections with certain goals in mind, we recommend items (or actions) that help users fulfilling their intended goals using their past only as a way of identifying goals of interest. We introduce a model that connects goals and actions through action sets implementing the respective goals. Such a model captures latent associations among goals and actions and allows the ranking of actions considering different user strategies such as to complete at least one goal with the minimum effort (i.e., minimum number of actions), or to open up more paths for fulfillment of more goals in the future. For each strategy we recommend an algorithm that exploits the user action and goal spaces to rank the actions in a different way. We have performed extensive experimental studies to understand how these techniques are related and compare the results against traditional recommendation methods. The experiments illustrate that it is not possible to replicate the results of our approach using existing techniques.

1 INTRODUCTION

People are daily facing situations in which they have to make choices from large collections of items. Selecting the best answer to a search engine query among those satisfying the query conditions, selecting a movie to watch, an item to purchase, or friend activities to read about in social media, are only some of the most characteristic examples. Recommender systems [3, 7, 12, 16, 20] give advice to users on items that are likely of interest to them. There are two main categories of recommender systems. The first is the collaborative filtering, which is based on the idea that similar users have similar preferences, thus, the analysis of the choices of similar users can result in successful recommendations of items that have not been selected yet. The second category is the content-based which is based on the idea that users would like items that have similar features with items they have liked in the past. The principle behind both approaches is that whatever the past indicated as preference, it is likely to be preferred also in the future.

In this work, we approach the problem based on a different principle. There have been studies in psychology and social sciences [4] that have shown that human actions are not random and unrelated events. They may be of course affected by preferences but they are mainly results of rational selections performed with

the purpose of achieving some specific goal that a person has set and aims to fulfill [1].

Based on these studies, we advocate that by recognizing the goals for which actions of the past have been performed, it is possible to identify the driving forces of the users' future actions and make recommendations that better fit these needs. Since the fulfillment of a specific goal may require actions that are highly different in nature, this form of recommendation may recommend actions that are highly different from those of the past, or from those that similar users have done in the past. Note that we may use the term "actions" and not "items" as typically done in recommender systems; with this option we are being more generic since the selection of an item, the purchase of a product, or the watching of a movie are practically all actions.

Existing studies in recommender systems have already recognized that methods taking into account similarity with what has happened in the past are not always matching user expectations and have tried different techniques that focus on other aspects such as serendipity, novelty and diversity to improve the quality of recommendations [9]. However, these solutions are not principled and are not driven by some specific, user-selected, well-defined target while in many recommendation scenarios there exist targets that users are willing to reach. For instance, in online learning platforms, users may target at specializations or/and degrees. In employment-oriented social networking services such as LinkedIn users are encouraged to take actions that will lead them into their next position. In addition, they can see how some actions can lead to the same target following different career paths. Moreover, users may perform actions that will lead them to the fulfillment of commercial goals such as to get discount coupons, or everyday goals such as to become fit or to cook.

Consider, for instance, the case of a customer in a supermarket that has placed in the cart a kilo of potatoes and carrots. A content-based recommendation will try to propose products that are close to what is already in the cart, i.e., similar to potatoes and carrots which means it may propose other kinds of vegetables, or even suggest other types of potatoes. On the other hand, a collaborative filtering system may suggest light beer or red peppers, because these items have been bought in the past by customers with similar preferences. Both methods, through clearly different routes, recommend items based on the customer's past. Instead, by taking into account that the items in the customer's cart can be combined with other items to produce one or more food recipes, the system can open up new options to the customer. For instance, considering a recipe to make an olivier (russian) salad that includes: potatoes, carrots and pickles, an item to be recommended would be pickles. Another useful ingredient would be nutmeg that is a spice used for mashed potatoes and pan-fried carrots, two recipes that require products some of which are already in the customer's cart. Such a recipe-based recommendation of products may not be justified by similarity to products already in the cart, neither by other product combinations found frequently in the

carts of other customers. This means that neither association rules nor techniques that detect correlations among items can be employed to make such recommendations since they highly depend on the popularity of these item sets. So, unless we consider the product combinations found in the recipes, these products will not be recommended by other techniques. Furthermore, given the recipes, the recommendations can be optimized for an overall benefit. For example, recommended products may give the ability to the customers to maximize the number of recipes that they can materialize.

Considering goals in the recommendation problem is challenging. The challenge comes from the fact that, in real life, there are typically multiple goals that one needs to fulfill at any given time. Each of these goals may require fewer or more actions in order to be fulfilled, and there may exist alternative ways for the fulfillment of a specific goal. Users have to reason on the priorities between the goals they try to achieve and the benefit they will have by the execution of each action towards the fulfillment of these goals. For instance, some users may prefer actions that help them fulfill a goal as soon as possible, while others may prefer actions that help the advancement of as many goals as possible. A goal-oriented recommender will have to leverage the goals by first recognizing the intended user goals, decide the priorities among them, and quantify the benefit of each action in relationship to the intended goals and in conjunction with the other possible actions.

We introduce a new family of recommendation strategies, i.e., *goal-based recommendations*, that deal with the above challenges. The goal-based strategies identify the goals for which exists evidence that the user is aiming at achieving. The evidence originates from the previous user activity, i.e., the actions that the user has already performed. Given this goal space, the strategies explore the sets of actions that lead to the fulfillment of these goals and contain actions that the user has already performed to find actions which the user has not performed and may be willing to complete. The sets of actions together with the goals they fulfill constitute the user's *goal implementation space*. The likelihood that the users will like an action from the candidate set of actions in this space depends on their approach towards the goals they would like to fulfill. We have identified three different strategies for exploring and exploiting the user's spaces in order to select the actions to be recommended. The three strategies correspond to three different policies based on which users often make their selections.

The first strategy is the *Focus* that examines each of the action sets in the user's goal implementation set to find which of them lead to the fulfillment of the goal that is closest to completion, either because most of the required actions have been already performed (*Focus_{cmp}*), or because they require only a few more actions (*Focus_{cl}*). Then, it forms the recommendation lists from the actions in these action sets. It is the policy preferred by users that need to fulfill at least one goal through the actions in the current recommendation list. The second strategy, *Breadth*, is not examining each action set in the user's goal implementation space separately. It considers more than one set of actions at the same time. Specifically, it evaluates and ranks the actions in the user's action space based on all the sets this action participates and selects those actions that belong in as many sets as possible together with as many as possible actions from the user activity. This strategy is for users that would like to fulfill as many goals as possible, if possible, through this recommendation list, but in order to maximize the number of fulfilled goals, they are willing to complete some or all of them in the future, i.e., not only through the actions in the current recommendation list. This way it keeps some "paths"

open for the future (i.e., unfulfilled goals) but those paths contain the minimum number of additional actions. We also suggest a third strategy, the *Best Match*, that similarly to *Breadth* is not trying to fulfill at least one goal through the current recommendation list. It recommends actions that contribute to the goals of the user's goal space. However, in contrast to *Breadth*, *Best Match* evaluates an action considering the whole goal space, not only the goals to which this specific action contributes. It generates a profile for the user and estimates a similarity between this profile and the actions to be recommended. The action representation shows how much that action contributes to the fulfillment of the various goals and the user profile how many of the user actions contribute to the various goals. It is a policy that may end up in the fulfillment of many goals in the future. However, it is a strategy for users that are interested in actions that are more useful (contribute more) to the goals to which the user has put more effort in the past (and respectively less to goals to which the user has put less effort).

Our contributions can be summarized as follows:

- We introduce and formally define the notion of goal-oriented recommendation, which evaluates every action considering the goals which the current user may be willing to fulfill and how that action contributes to the fulfillment of one or more of these goals together with other actions of the user (Section 3).
- We explain how it differs from existing techniques and why the latter cannot be used to offer this type of recommendation (Section 2).
- We present different strategies for ranking the candidate actions, with each strategy implementing a different policy in prioritizing the goals and selecting the actions to be recommended (Section 5).
- We describe efficient ways of implementing the above strategies and materializing the goal oriented recommendation paradigm (Section 4).
- We study the effectiveness of our methods and compare them to the state-of-the-art recommendation approaches. We show that goal-based approaches can recommend actions that bring the user closer to the fulfillment of goals that are related to her/him, are highly different from each other and at the same time from actions performed by other users in the past (Section 6).

2 RELATED WORK

Goal modeling. Goal modeling has attracted a lot of research interest for decades. However, the focus of the different fields has been in goal and next action inference [13] such as prediction of the next action in a sequence, e.g., the next web page to click or the next location to be [11, 18]. The purpose of such systems is for instance to promote the inferred actions or act in anticipation of the user's actions [2]. To infer the next action(s) they employ models such as probabilistic (state transition) models, e.g., Bayesian Networks [15], or Markov models [18] or other variations [2].

Recommender Systems. Our method retrieves actions to be recommended but does not consider neither the user's neighborhood activity nor the activity of the current user as in state-of-the-art recommendation approaches but the actions in the implementations of the various goals. In contrast to *Collaborative Filtering* [7, 8] that exploits previous item selections or interactions that similar users have performed, it selects implementations that contain subsets of the user activity and can be adequately extended to lead to

the fulfillment of one or more goals. It also differs from *Content-based Filtering* [3] that recommends items similar to what the user has used in the past with a high degree of satisfaction.

Association rule mining. Association rule mining analyzes the user’s histories to identify groups of items appearing together and use this as the basis for making recommendation [19]. The approach is based on popularity, while our technique is not affected by popularity fluctuations. Furthermore, different actions may often appear together but for different goals, which means not only that recommendations different than ours will be made, but these recommendations will also be incomplete, i.e., they will manage to fulfill none of the goals, since the system is confused and unable to distinguish the different intended goals of the actions that appear together.

3 GOAL-BASED RECOMMENDATIONS

Actions, Goals and Goal Implementations. We assume the existence of a set \mathcal{U} of users. Users perform actions such as the purchase of an item, the visit of a web page, the watching of a movie, or any other recordable task. We consider the existence of a set \mathcal{A} of actions.

People set the goals that they need to achieve and then they decide to perform those actions that they believe will help them fulfill their goals. We denote \mathcal{G} the set of goals. A set of actions constitutes an *activity*, which means that there are $2^{|\mathcal{A}|}$ different possible activities. The activities that are intended for a goal $g \in \mathcal{G}$, alongside the respective goal, are referred to as *goal implementations*.

Definition 3.1. A *goal implementation*, or simply implementation, is a pair $\langle g, A \rangle$ with $A \subseteq 2^{|\mathcal{A}|}$ and $g \in \mathcal{G}$.

Goal Implementation Data sources. Goal implementations can be found in sources related to almost every aspect of human activity. Recipes, for instance, are implementations of specific goals (the food that the recipe is about). Online learning platforms have specializations and degree that are implemented through courses. Each specialization is associated with one or more set of courses indicating the actions required to achieve the goal, i.e., the specialization. Goals can also be found in online stores. Many online clothing stores, for instance, give users the ability to form *outfits* and annotate them with labels such as “for friend meetings”, “to be warm” and so forth. Those outfits constitute implementations of the goal, with the goal being the label. With this knowledge, when the system recommends some item to a user apart from considering the user preferences on characteristics such as color or material (content based filtering) or considering what clothes others have bought in the past (collaborative filtering), it can employ a goal-based recommendation technique and suggest items that can be combined with clothes the user has bought in the past to form complete outfits.

Another rich source of goal implementations are social networks or specialized web sites where users record and share success stories of things that they do in life. Examples include 43Things (<https://43things.com>) and wikihow (www.wikihow.com), where users describe actions to achieve real-life goals. There are many works on transforming such textual descriptions into a structured form, like an ontology [10, 14, 17], or a taxonomy [6, 21]. They typically employ structural information such as HTML tags or enumeration. A different way to create such datasets from web pages is by posing queries of the form “in order to + a goal description” on search engines [21].

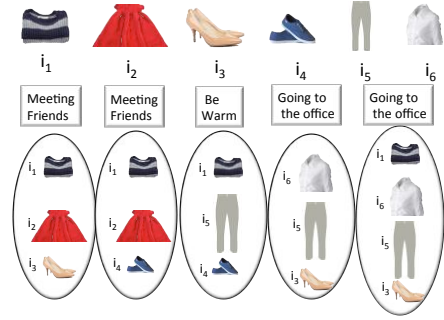


Figure 1: Combinations and the goals they serve

One of the datasets that we are using in the experimental evaluation of this work contains 18k goal implementations that we have extracted by performing *action identification* on user-generated descriptions about everyday goals such as *learn english*, *travel to Italy* and so forth from the 43Things website. We did this action extraction with a module that we have developed for this purpose, that works on a simpler model and for plain text (ref. Section 4). We do not elaborate further on the extraction task since it is orthogonal to the focus of the current paper and a different line of work of ours.

Example 3.2. Figure 1 depicts a set of goal implementations from an online clothing store. We denote a goal implementation set as L . The columns indicate outfit purposes (the goals) while the rows are the items (the actions). If we depict by a_k the action of buying the item i_k , then the implementation set is:

Implementation	$\langle \text{Goal}, \text{Activity} \rangle$	
$p1$	$\langle g_1, A_1 \rangle$	where $A_1 = \{a_1, a_2, a_3\}$
$p2$	$\langle g_2, A_2 \rangle$	where $A_2 = \{a_1, a_2, a_4\}$
$p3$	$\langle g_3, A_3 \rangle$	where $A_3 = \{a_1, a_4, a_5\}$
$p4$	$\langle g_3, A_4 \rangle$	where $A_4 = \{a_3, a_5, a_6\}$
$p5$	$\langle g_3, A_5 \rangle$	where $A_5 = \{a_1, a_3, a_5, a_6\}$

Recommendation Setting We assume an implementation set L . The set may have been constructed through one of the many methods mentioned earlier that are already available in the literature, or through the text-based module we developed for the experimental evaluation.

The actions that the user has already performed is referred to as the *user activity* H . We do not know why these actions have been performed but given the goal implementation set, there is a number of possible goals that the user may have had in mind when performing each of these actions. These goals constitute the *goal space* of the activity H .

The goal space makes all the actions that contribute to one or more goals in the goal space to be likely of interest to the user. Our aim is to recommend to the user actions that are not in H , and which the user would be happy to perform. However, not all the actions offer the same benefit. What action the user would be more willing to perform depends on what priorities the user puts on the goals. Some actions may help towards the fulfillment of many goals, while others towards the fulfillment of goals almost completed. Thus, we need to create a ranked list of the actions to recommend according to some criterion. Depending on the criterion/policy we use, a different recommendation strategy is materialized. These policies comprise the topic of the next section.

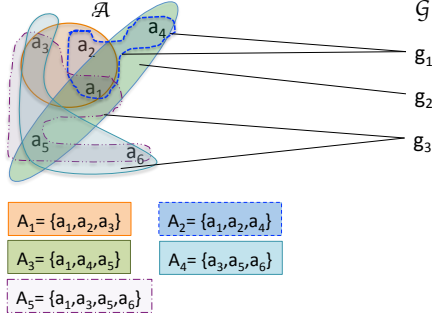


Figure 2: Illustration of our model.

4 GOAL MODEL

Our aim is to recommend to the users a set of actions considering the goals that they can fulfill given their activity. Those goals are associated with at least one action from the user activity, i.e., at least one action contributes to one or more of their implementations. An action $a \in A$ is said to *contribute* to a goal g through a goal implementation p , when there exists a goal implementation $p = (g, A)$. and we denote it as $a \rightsquigarrow^P g$.

The set of goals that are associated with an action forms its *goal space*.

Definition 4.1. Given a goal implementation set L , the *goal space* of an action a is the set $\mathcal{GS}(a) = \{g \mid g \in \mathcal{G} \wedge p \in L \wedge a \rightsquigarrow^P g\}$.

The goal space extends naturally to the case where we have a set of actions A instead of one, to be the union of the goal spaces of the individual actions in A , i.e., $\mathcal{GS}(A) = \bigcup_{a \in A} \mathcal{GS}(a)$. Considering the individual goal spaces of each action $a \in H = \{a_1, \dots, a_n\}$ there are two extreme cases: (i) $\mathcal{GS}(a_1) \cap \dots \cap \mathcal{GS}(a_n) = \emptyset$ (i.e., there are no common goals) and (ii) $\mathcal{GS}(a_1) \cap \dots \cap \mathcal{GS}(a_n) = \mathcal{GS}(H)$ (i.e., the goal spaces of all actions are the same), where we have no evidence whether one or more goals constitute a priority for the user. In all the other cases the goal spaces are a valuable source of information and should be exploited for retrieving the actions.

Another important factor that should be considered is that actions are not independent from each other since subsets of actions co-contribute to one or more goals. Therefore, given an action, there exist other actions that should be also performed in order for a goal in the goal space to be fulfilled. The set of these actions forms the *action space* of an action.

Definition 4.2. Given a goal implementation set L , the *action space* of an action a is the set $\mathcal{AS}(a) = \{a' \mid g \in \mathcal{G} \wedge p \in L \wedge a \rightsquigarrow^P g \wedge a' \rightsquigarrow^P g \wedge a \neq a'\}$.

The action space, similarly to the goal space, extends naturally to the case where we have a set of actions A instead of one, to be the union of the action spaces of the individual actions in A , i.e., $\mathcal{AS}(A) = \bigcup_{a \in A} \mathcal{AS}(a)$.

Example 4.3. In the implementation set of Example 3.2, since action a_1 participates in the activities A_1, A_2, A_3 and A_5 , its implementation space is the $\mathcal{IS}(a_1) = \{p_1, p_2, p_3, p_5\}$, and its goal space the $\mathcal{GS}(a_1) = \{g_1, g_2, g_3, g_5\}$. Its action space is the set of all the other actions in A_1, A_2, A_3 and A_5 , i.e., $\mathcal{AS}(a_1) = \{a_2, a_3, a_4, a_5, a_6\}$.

Thus, above we have determined two very important association types that correspond to two basic “operations” given an activity, i.e., a set of actions A : to form the *goal space* $\mathcal{GS}(A)$, and

the *action space* $\mathcal{AS}(A)$. Moreover, we need a matching function connecting the goals with the action set(s) that implement them.

We suggest a model that sees each activity A in the L as a *hyper-edge* that connects the actions that participate in it. Moreover, it labels each activity A with the goal that fulfills given a goal implementation (g, A) . Figure 2 graphically illustrates our model that we call *association-based goal model*.

Given a small goal implementation set, we can, for instance, form the user goal space by visiting one by one all the implementations and check whether there exist any common actions in the user activity and their activity (i.e., the set intersection). However, when moving to hundreds or millions of implementations, the cost gets prohibitive. Therefore, we should implement our model in a way that apart from capturing all the associations, *allows us to form efficiently the goal and action spaces considering the interconnections among actions and goals through the goal implementations*.

In order to retrieve the information we need in real time, we employ a set of indexes. We first build an index A -*idx* for the action set and an index G -*idx* for the goal set. Keeping the information derived from the goal implementation set L needs a more complex structure. We refer to each goal implementation using a unique identifier *id*. We split the information of the goal implementation pairs in two indexes: *Goal Implementation ActiVity index (GI-AV-idx)* and *GI-G-idx (Goal Implementation Goal Index (GI-G-idx))*. The first one matches the activity of a goal implementation to the *id* of the goal implementation where it belongs. We store a set with the *ids* of the actions. The second index matches each goal *id* to all the implementation *ids* that exist for the specific goal. Now we need to connect the goal implementations with the actions they contain. For this, we use A -*GI-idx (Action to Goal Implementation Index)* that retrieves all the goal implementation *ids* where an action contributes, i.e., the implementation *ids* (*pIds*) s.t., $a \rightsquigarrow^P g$.

Equations 1 and 2 describe how we exploit the above index structures to implement the basic operations that we described earlier, i.e., to form the goal and action space given an activity.

$$\mathcal{GS}(A) = \{GI-G-idx[pId] \mid a \in A \wedge aId = A-idx[a] \wedge pid = A-GI-idx[aId]\} \quad (1)$$

$$\mathcal{AS}(A) = A - \{GI-AV-idx[pId] \mid a \in A \wedge aId = A-idx[a] \wedge pid = A-GI-idx[aId]\} \quad (2)$$

5 STRATEGIES

Having built the *association-based goal model* described above, we can retrieve the actions to be recommended exploiting their associations with the actions in the user activity and the goals in the user goal space. In practice, we perform set operations to evaluate how strong the associations are. We suggest different strategies considering options with which we believe users prioritize actions. The first strategy examines the associations of the user activity and each of the sets of actions that contribute to goals from the user goal space. Examining each set of actions separately and not in conjunction with other action sets as well helps users to stay *focused* on one goal at a time (*Focus* strategy). On the other hand, the second strategy examines more than one action sets at the same time. For this reason we call it *Breadth*. Breadth gives priority to actions that are strongly associated with each other and the user activity; such actions can be exploited in the fulfillment of a subset of the user goal space at the same time. There is

also a third strategy that considers at the same time all the action sets that are associated with the user activity. In this case, the associated goals are an evidence of the user preferences on goals (goal-oriented profile). The strategy is referred to as *Best Match*. The three strategies that are called respectively *Focus*, *Breadth* and *Best Match* are described in the Subsections 5.1 and 5.2, 5.3 respectively.

5.1 Focus

Strategy *Focus* gives the user the option to have access to actions that lead to the *completion* of one of the goals in the user goal space. For an action set A in a goal implementation $\langle g, A \rangle$ where $g \in \mathcal{GS}(H)$, the intersection $|A \cap H|$ gives the number of actions in the implementation that have been already performed. Focus does consider the association of the user activity and the actions sets in the implementation set. But that is not enough to retrieve the actions that together with a subset of the user activity H form the activity of a goal implementation in the library L , i.e., they comprise the actions that are required for the goal to be completed. For this purpose we introduce two measures, goal implementation *completeness* and *closeness*, that evaluate and rank the candidate action sets and by extension the respective goal implementations. Completeness considers the proportion of the actions that are common in the user activity (set intersection) and the actions in the examined action set while closeness considers the common actions in comparison with the remaining actions.

$$completeness(\langle g, A \rangle, H) = |A \cap H|/|A| \quad (3)$$

$$closeness(\langle g, A \rangle, H) = 1/|A - H| \quad (4)$$

Goal implementation completeness is inspired by plan inference in plan libraries for intelligent agents [5]. However, in intelligent agents the aim is to predict which sequence of actions the agent is following (i.e., the agent has already selected a plan) while in our problem the recommendation mechanism aims to guide the user to options that s/he may have not considered without the recommendation system.

Algorithm 1 Focus Ranking

Ranks actions based on completeness (step 3) or alternatively closeness (step 3) of the corresponding goal implementations*

Input Set H , Set CA , int k

Output List R

1 $CI \leftarrow [], R \leftarrow \{\}$

//get the goal implementations that connect the goals in $\mathcal{GS}(H)$ with the actions in H

2 *for all* $aId \in H$, *for each* $pid \in A-GI-idx[aId]$

3 $IS = \{ A-GI-idx[aId] \mid a \in A \wedge aId = A-idx[a] \}$

4 *end for all*

5 *for each* pid in IS

6 $\langle p, sc \rangle \leftarrow \langle pid, \frac{|A \cap H|}{|A|} \rangle$

6* $\langle p, sc \rangle \leftarrow \langle pid, \frac{1}{|A-H|} \rangle$

7 $CI.add(\langle g, A \rangle, sc)$

8 *end for all*

9 rank CI based on sc

10 return the top k actions from the action sets of the top implementations (set CI)

We rank the goal implementations in the set of the implementations that are associated with the user activity in descending

order of completeness. Given this ranking, the list of action recommendations is formed as follows. We pull from the first goal implementation all the actions that have not been performed yet, i.e., that are not in the user activity, and we add them to the recommendation list. If more actions are needed for the top- k list, we pull the next goal implementation and so forth, until the list gets full. Note that it may be the case that the remaining slots in the top- k list are fewer than the actions of the next goal implementation in the ranked list of implementations. In this case, we can decide to leave the list with fewer recommendations or expand k to include the required actions for this implementation.

The completeness ranking function promotes the actions in the activities of the goal implementations with the largest completeness (see Algorithm *Focus Ranking*, line 3). This way the recommendation mechanism guides the user to actions that will lead to the fulfillment of the goal for which the user has already done most of the work, i.e., she has performed most of the actions needed for its fulfillment.

5.2 Breadth

With the strategy *Focus*, a single goal drives the recommendation process. This can be very restrictive if the user is not that determined to fulfill the specific goal. Therefore, we give the user another option: *Breadth* that evaluates every candidate action a considering a subset of goals in the goal space. This subset consists of the goals that are connected to the candidate action a through one or more goal implementations, i.e., the goal space $\mathcal{GS}(a)$. The reasoning behind this is that since every action in the action set \mathcal{A} can participate at the same time in more than one goal implementations in the set L and possibly contribute to a number of goals, its benefit should be estimated based on all these goals.

First, in order to evaluate a candidate action a , we should take into consideration the number of goal implementations in its implementation space, i.e., the $IS(a)$. We will refer to this quantity as *utility*.

$$u(a) = |\{A - GI - idx[aId] \mid a \in A \wedge aId = A - idx[a]\}| \quad (5)$$

$\forall pid \in A-GI-idx[aId]$. The larger the utility of an action, the larger the benefit that the user can have by a single action. For instance, in the Example 3.2, the action of buying item i_5 (i.e., a_5) is part of three goal implementations: p_3, p_4, p_5 , i.e., it can be used in 3 different outfits. Hence, it can be considered as more beneficial to the user compared to the action of buying i_6 (i.e., a_6) that contributes only through 2 goal implementations: p_4 and p_5 . However, considering the user activity $H = \{a_2, a_3\}$, we remark that the user has not showed interest to goal implementation p_3 ($p_3 \notin IS(H)$). Consequently, goal implementation p_3 should not have been taken into consideration. Thus, we need a measure that captures the utility of an action considering the user activity as well. Moreover, recommending actions of high utility is not enough. We should also consider how related, or else strongly connected, is a candidate action to the user activity. To do so, we need to consider how many of the actions in the user activity are connected to action sets that fulfill goals in the examined goal subspace.

$$sc(a, H, Breadth) = \sum_{\forall \langle g, A \rangle \text{ where } A \cap H \neq \emptyset, \text{ and } a \in A} |A \cup H| \quad (6)$$

The above equation captures both the utility of a candidate action and its relatedness to the user activity. Now, we can rank

Algorithm 2 Breadth Ranking

Ranks Candidate actions based on all the Implementations of the user's Implementation space where they participate

Input: Set H (user activity), int k

Output: top k actions

```
1   $R \leftarrow \{ \}$ 
2  for all  $aId \in H$ , for each  $pid \in A\text{-GI-idx}[aId]$ 
3   $\mathcal{IS} = \{ A\text{-GI-idx}[aId] \mid a \in A \wedge aId = A\text{-idx}[a] \}$ 
4  for each  $pId$  in  $\mathcal{IS}$ :
5     $ActionsInP \leftarrow GI\text{-AV-idx}[pId]$ 
6     $comm \leftarrow |ActionsInP \cap H|$ 
7    for each  $aId \in ActionsInP$ 
8      if  $aId$  in  $R.keys$ :
9         $R.add( \langle aId, R[aId] + comm \rangle )$ 
10     else:
11        $R.add( \langle aId, comm \rangle )$ 
12  rank  $R$  on score value and return the top  $k$  actions
```

the candidate actions and get the recommendation list R . To form the recommendation list, we rank the candidate actions using the function described in Equation 6.

Algorithm *Breadth Ranking* presents in pseudocode the steps of the *Breadth*. The algorithm does not estimate the score (ref. Eq. 6) of each action in the $\mathcal{AS}(H)$ separately. It examines each associated implementation and updates the score of all the actions of the $\mathcal{AS}(H)$ that belong in the current implementation. This way, when all the implementations have been examined the action scores (ref. Eq. 6) are ready and the action ranking takes place.

5.3 Best Match

Best Match policy in contrast to *Breadth* that evaluates each action in the user action space considering only the goals in the goal space to which this specific action contributes, considers the whole goal space. In fact, it generates a user profile that reflects the effort that the user has made towards all the goals and retrieves actions that contribute similarly to those goals. *Best Match* considering the user goal space, represents every action as a vector and aggregates the representations of the individual actions in the user activity into a single vector. The final vector constitutes the user profile. Subsequently, the candidate actions can be ranked based on their similarity to the user profile. Such an approach promotes actions that contribute to most of the goals in the user's goal space.

Goal-based user representation. In recommendation systems, user profiles are described in terms of the features of the items that a user prefers. In our case, a profile captures the user dedication towards a set of goals. We consider that the more actions from the user activity H contribute to a specific goal in the goal space $\mathcal{GS}(H)$ of the user activity, the more the user cares for this goal.

Hence, we consider that an action a can be represented as a vector \vec{a} in the feature space $F^{\mathcal{GS}(H)}$ (as an item is represented by considering features in content-based recommendation methods). One option would be to form a boolean vector, $\forall i \in \{0, |\mathcal{GS}(H)|\}$, where $g \leftarrow F^{\mathcal{GS}(H)}[i]$:

$$\vec{a}[i] = \begin{cases} 1, & \text{if } \exists p \leftarrow \langle g, A \rangle \text{ s.t. } a \in A, g \in \mathcal{GS}(H) \\ 0, & \text{otherwise} \end{cases} \quad (7)$$

The problem with the above representation is that it disregards the fact that an action in the user activity may contribute to a goal through one or more implementations. Therefore, instead of the boolean representation, we adopt a vector representation where $\vec{a}[i]$ is defined to be the number of goal implementations p s.t.

Algorithm 3 Get-Goal-Based-Profile

Creates the user profile that reflects her connections with the Goal Space

Input: Set H (user activity)

Output: \vec{H} vector in $\mathcal{GS}(H)$ that aggregates the contribution of all actions in H

```
1   $\vec{H} \leftarrow \emptyset$ 
2   $GP_{map} \leftarrow \emptyset$ 
3  for all  $aId \in H$ , for each  $pid \in A\text{-GI-idx}[aId]$ 
4   $\mathcal{IS} = \{ A\text{-GI-idx}[aId] \mid a \in A \wedge aId = A\text{-idx}[a] \}$ 
5  for each  $pId$  in  $\mathcal{IS}$ :
6     $gIdTmp \leftarrow GI\text{-G-idx}[pId]$ 
7    if  $gIdTmp$  in  $GP_{map}.keys$ :
8       $GP_{map}[gIdTmp] \leftarrow GP_{map}[gIdTmp] + 1$ 
9    else:
10      $GP_{map}[gIdTmp] \leftarrow 1$ 
11  /*convert map  $GP_{map}$  to a vector in  $F^{\mathcal{GS}(H)}$  space*/
12  for each  $gId$  in  $\mathcal{GS}(H)$ 
13     $\vec{H}.add(GP_{map}[gId])$ 
```

$a \rightsquigarrow^p g$ and $g \in \mathcal{GS}(H)$. The value in each position of the vector \vec{a} becomes $\forall i \in \{0, |\mathcal{GS}(H)|\}$, where $g \leftarrow F^{\mathcal{GS}(H)}[i]$:

$$\vec{a}[i] = \sum_{\forall p \leftarrow \langle g, A \rangle \text{ s.t. } a \in A, g \in \mathcal{GS}(H)} 1, \quad (8)$$

To get the *user profile*, we aggregate all the representations of the actions of the user activity in the feature space $F^{\mathcal{GS}(H)}$ into a single vector. The user profile captures for each goal in $\mathcal{GS}(H)$ how many of the user actions contribute to this goal considering the different goal implementations for the same goal as well. Since the user profile is generated based on the current user activity H , we denote it as \vec{H} .

$$\vec{H} = \sum_{\forall a \in H} \vec{a} \quad (9)$$

For example, for the user activity: $H = \{a_2, a_3\}$, the number of goal implementations where at least one of the actions of the user activity participate is 4. The user profile is $\vec{H} = \{3, 0, 2\}$. In the user profile is reflected the fact that the user has performed a_1 and a_3 that contribute to g_1 : “meeting friends” 3 times and to g_3 : “going to the office” via one goal implementations each, and that the user has shown her/his preference to the goals g_1 and g_2 over the rest of the goals in the goal space $\mathcal{GS}(H)$.

Goal-based representation of candidate actions. To rank the candidate actions against the user profile, we represent each candidate action in the same goal space, i.e., as goal vectors in the space $F^{\mathcal{GS}(H)}$ in the exact same way the actions from the user activity have been represented (ref. Eq. 8).

Distance-based Ranking. To rank the candidate actions, we can use a standard metric between the user profile and each of the candidate actions, as follows:

$$sc(a, H, \text{Best Match}) = dist(\vec{H}, \vec{a}) \quad (10)$$

For instance, considering the Example 3.2, action a_1 from the user activity H would be closer (smaller distance) to the user profile than that of a_4 since the first contributes to g_1 : “meeting friends” via two goal implementations and via another goal implementation to g_3 : “going to the office” as well; while the latter contributes to g_1 via only one goal implementation and to g_2 : “be warm” to which the user has shown no interest.

Algorithm 4 Best Match Ranking

Ranks actions based on their distance to the goal-based user profile

Input user activity H , int k

Output top k actions

```
1  $R \leftarrow \{\}$ ,  $CA \leftarrow \mathcal{AS}(A)-H$ 
2  $\vec{H} \leftarrow \text{Get-GoalBased-Profile}(H)$ 
3 for each  $aId$  in  $CA$ :
4    $GP_{map} \leftarrow \emptyset$ 
5    $IS \leftarrow IS(aId)$ 
6   for each  $pId$  in  $IS$ :
7      $gIdTmp \leftarrow GI-G-idx[pId]$ 
8     if  $gIdTmp$  in  $GP_{map}.keys$ :
9        $GP_{map}[gIdTmp] \leftarrow GP_{map}[gIdTmp]+1$ 
10    else:
11       $GP_{map}[gIdTmp] \leftarrow 1$ 
12    /*convert map  $GP_{map}$  to a vector in  $F\mathcal{GS}(H)$  space*/
13    for each  $gId$  in  $\mathcal{GS}(H)$ 
14       $\vec{a}.add(GP_{map}[gId])$ 
15     $\langle aId, sc \rangle \leftarrow \langle aId, dist(\vec{a}, \vec{H}) \rangle$ 
16     $R.add(\langle aId, sc \rangle)$ 
17 rank  $R$  on  $sc$  and return top  $k$  actions
```

Algorithms *Get-GoalBased-Profile* and *Best Match Ranking* describe the procedure. *Get-GoalBased-Profile* forms the goal-based vector representation of the user (user profile) by considering for each action in the user’s activity all the implementations where the examined action belongs (i.e., its implementation space) in order to find to which of the goals of the user’s goal space it contributes and add one in the respective position of the vector \vec{H} . On the other hand, *Best Match Ranking* compares the user profile with the goal-based vector representation of each action in CA by considering again the goal implementation space of the actions and the goals to which they contribute. and ranks them according to their distance with the user profile to get the top k .

5.4 Complexity analysis of strategies

The complexity of the goal-based recommendation mechanisms is mainly determined by the action *connectivity* of the association-based goal model, i.e., the average number of implementations where an action belongs. *Focus* first retrieves all the implementations that are associated with the user activity and estimates the *closeness* or the *completeness* of each of the implementations. Therefore, the cost of the mechanism is estimated as the product of $|H|$, connectivity and of the cost of set intersection or asymmetric set difference that are the main operations of the two alternatives of the *Focus* algorithm. The estimation of completeness may be more time consuming in practice (ref. Figure 7). For instance, in our implementation, intersection takes more time than set difference for sets of equal size due to the larger number of element removals that are required in the latter. *Best Match* also starts with retrieving the implementations that are associated with the user activity and then transforms them to vectors in the feature space $F\mathcal{GS}(H)$ ($O(|H| * connectivity)$). Subsequently, this transformation is also performed in all the actions in the action space of the user activity $\mathcal{AS}(H)$. For an action a , the complexity of forming its space $\mathcal{AS}(a)$ depends again on the action connectivity and the average implementation length. Since connectivity is significantly higher than the size of the average goal implementation and the

user activity, the time cost is mainly determined by connectivity: $O(connectivity * |H| + connectivity * average\ implementation\ length)$. On the other hand, *Breadth* retrieves the associated implementations and gets the intersections of the actions in each of the implementations and the user history. The asymptotic time complexity for the first step is $O(|H| * connectivity)$ and for the second step $O(connectivity * set\ intersection\ cost)$.

In practice, if we consider two association-based goal models built on sets of the same connectivity with the second one containing more implementations, the recommendation time would be higher for the set with the larger number of implementations. The reason is that in larger implementation sets the spaces of associated actions, goals and implementations of the individual actions in the user activity are not overlapping and by consequence their union gets larger. Nevertheless, the algorithms scale very well even on sets of millions of implementations (ref. Figure 7).

6 EVALUATION

For our experimental evaluation, we examine two different scenarios: (a) a grocery store where clients (users) buy food products, and (b) a system where users record actions they perform in their lives such as *read a book* or *eat healthy*. We selected these scenarios to show that goal-based recommendation can be used both in practical scenarios where existing recommendation techniques have already been applied, and to offer innovative services that have not been available so far. In both scenarios, we want to recommend to the users *actions of interest* (i.e., buy + “a product” and everyday actions respectively). The actions are characterized to be of interest based on the goals that they serve: food products can serve *food recipes*, while everyday actions can serve *life goals* such as *lose weight* or *learn english*. Another reason for examining these scenarios is that they cover two different cases: the first one covers the case where the same action participates in a great range of goal implementations (on average 1.2K impl.), while in the second case, most actions are limited to specific “families” of goals (on average an action participates in 3.85 implementations).

Dataset Description. The first dataset is an open source grocery shopping dataset (<https://github.com/julianhyde/foodmart-data-mysql>) that contains 1560 *food products* (items) and records of customer purchases in different time slots, i.e., *carts*. The food products are organized in 128 (sub)classes such as “baking goods”, “seafood”, “fruit”, “spices” and so forth. Clients can utilize these products in various *recipes* to produce *different dishes* (goals). We used a dataset of 56.5k *recipes* from a food ontology (<http://data.lirmm.fr/ontologies/food#Recipe>). The number of implementations in which an action participates on average, i.e., the *connectivity*, is 1.2K. We run our recommendation techniques and the state-of-the-art algorithms using as input, i.e., current user activity, 20.5k client carts.

The second dataset consists of goal implementations from a *goal-setting online social platform* called 43Things where users could publish the goals they set in their lives, “cheer” other users’ goals and efforts, and provide descriptions about how they managed to fulfill their goals. We have extracted 18047 *goal implementations* that contain 3747 *goals* such *pay my debts*, *get a new job*, *lose weight*, and 5456 *actions* e.g., *stop eating at restaurants* and *drink more water*. Both goals and actions are identified by unique identifiers. In contrast to the foodmarket dataset, users are focused on a few real-life goals. In total, we have examined 8071 users. The majority of the users (5047 users) are pursuing one goal, 1806 of them pursue 2 goals, 623 pursue 3, and 595

Goal Id	Actions Performed for Goal Fulfillment
g_1	{3,4,5,6,7, 8,9,10,11}
g_2	{12,13,14,15,16,17,18,19,20}
g_3	{1,2}

Initial User Activity	{3,4,5,6,7, 8,9,10,11,12, 13,14,15,16,17,18,19,20,1,2}
User activity used as input (unhidden actions)	{11,20,8,12,1,13 }

Table 1: Forming of user activity.

pursue more than 3 goals. Moreover, the action *connectivity* is very low, 3.84. The fact that actions here in contrast to actions that involve food ingredients are useful in a narrow range of goals and by extension goal implementations makes the analysis of the two sets more intriguing. User activities consist of all the actions that a user has performed for the fulfillment of all the goals that s/he has set. Therefore, in order to evaluate the recommenders we should hide a portion of the actions from the real user activities before applying the recommendation techniques. For instance, Table 1 illustrates all the actions that a user has performed to fulfill three goals. To get the input for the recommenders, we first concatenate the actions of the three respective implementations into the vector {1, 2 . . . 20}. Subsequently, the vector elements are shuffled and the 30% of the actions is considered to be the known user activity. The rest 70% is kept for evaluation purposes. Specifically, the recommenders, having some evidence that the user is interested in fulfilling one or more goals, should retrieve actions that are associated with those goals. In the example, the activity that remains unhidden consists of 6 actions. Two of the actions regard the first goal, three of them the second goal, and one action regards the last goal. Actions are both about goals that are closer to fulfillment and about goals for which there is no strong evidence. Some goals may also remain hidden.

Comparison with the State-of-the-art. Beyond the goal-based mechanisms, we examine how state-of-the-art recommendation approaches behave under the same context. We consider a nearest-neighbor Collaborative Filtering method (CF KNN) [20] and a matrix factorization method that employs alternating least squares with weighted-lambda-regularization (ALS-WR) to factorize the user-item matrix before performing the recommendations (CF MF) [8]. For the CF KNN since the user feedback is implicit, i.e., user *selection*, *non-selection*, the neighborhoods have been formed by employing jaccard coefficient, or else Tanimoto coefficient. The used implementation of the CF MF is from Mahout framework (<https://mahout.apache.org/>). We also consider a Content-based method that represents actions and users using domain-specific features, i.e., that builds vector-based representations for profiling the actions and users. For the foodmarket dataset the used domain-specific features are the 128 (sub)categories of the food products (e.g., “baking goods”, “seafood”). Based on the description of each food product, we matched each product to an ingredient leaving out products that are not included in any recipe, such as napkins. Therefore, each cart can be seen as the *user activity*, the set of recipes as the *goal implementation set L*, while the *actions* refer to the purchase of certain products/ingredients. On the other hand, for the 43T dataset, there are no widely accepted domain-specific features; therefore, we do not apply the content approach. For the goal-based recommendations, we used the methods described in Subsections 5.1, 5.2, and 5.3), namely *Focus_{cmp}* and *Focus_{cl}*, *Breadth* and *Best Match*.

Subsection 6.1 compares all methods. Subsection 6.2 focuses on the time efficiency of the goal-based methods.

6.1 Evaluation and Comparisons

Since we are introducing a novel recommendation approach, in Subsection 6.1.1, we first verify that this approach, indeed offers a different perspective to the users. To do so, we perform several *comparisons* on the results (i.e., the recommendation lists) produced by all the goal-based and the standard recommendation mechanisms.

- We compare the lists formed by the goal based mechanisms with the two standard recommendation mechanisms (ref. C.1.1. *Result Overlapping*).
- Collaborative filtering is based on the past activities of similar users (to the current user), while our algorithm is intended for discovering useful actions, i.e., actions that will help the user fulfill one or more goals. Thus, we examine whether actions that appear frequently *in the activities of other users* (popular actions) appear frequently *in the recommendation lists* as well. In other words, we study which recommendation mechanisms perpetuate the collective user behavior (ref. C.1.2. *Correlation of appearances in the user activities and the respective recommendation lists*).
- Next, we examine how useful the actions in the top-10 lists of each algorithm are for the user. To measure *usefulness*, we estimate the completeness of the goals in the user’s goal space after s/he performs the recommended actions (ref. C.1.3. *Usefulness*).
- We also study how *similar* the recommended actions in each list are presenting their (max, min and avg) pairwise similarity based on their domain-specific characteristics. Retrieving items that are very similar to each other is often considered a drawback of the Content-based filtering. It is important to understand how the rest of the examined approaches work as well (ref. C.1.4. *Pairwise similarity of the recommended actions*).
- Moreover, we examine how many of the actions in the recommendation lists have been indeed performed by the users. These actions are not of course part of the considered user activity but the users “like” them since they have performed them at some point (ref. C.1.5. *Average Percentage Of Recommended Actions that the user has indeed Performed*.)

Subsequently, Subsection 6.1.2 further examines the actions retrieved by the goal-based mechanisms (ref. C.2.1 *Frequency of Retrieved Items*) and presents the percentage of common actions in their top-10 recommendation lists (ref. C.2.2 *Result Overlapping of Goal-based methods*).

6.1.1 Comparison of all Approaches. C.1.1. Result Overlapping. Table 2 illustrates a very low overlapping of the top-10 lists formed by the goal-based mechanisms with the lists formed by the two state-of-the-art approaches. This result is expected, since as we have explained in Section 2, these approaches adopt fundamentally different philosophies.

C.1.2. Correlation of the number of appearances in the user activities and the number of appearances in the respective recommendation lists of the top-20 most popular actions. Table 3 illustrates the Pearson’s correlation between these two numbers. Correlation

Methods	Food Market			43T		
	Overlap.with Content Filt.	Overlap.with Collab. Filt. kNN	Overlap. with Collab. Filt. Matrix Factorization	Overlap. with Collab. Filt. kNN	Overlap. with Collab. Filt. Matrix Factorization	
Best Match	2.31%	0.85%	0.34%	0.13%	0.06%	
Focus _{cmp}	1.49%	0.85%	0.36%	0.11%	0.008%	
Focus _{cl}	1.85%	0.85%	0.35%	0.08%	0.01%	
Breadth	2.32%	0.86%	0.35%	0.26%	0.11%	

Table 2: Overlap of the top-10 actions retrieved by the goal-based mechanisms and the standard recommendation approaches.

Methods	Food Market	43T
	Correlation	Correlation
Content	0.115	-
Collaborative KNN	0.45	0.75
Collaborative MF	0.78	0.87
Best Match	-0.13	-0.24
Focus _{cmp}	-0.048	-0.26
Focus _{cl}	-0.02	-0.27
Breadth	-0.04	-0.15

Table 3: How correlated the recommendation lists with the top-20 popular actions in the user activities are.

Methods	Food Market Completeness			43T Completeness		
	Avg-Avg	Avg-Max	Avg-Min	Avg-Avg	Avg-Max	Avg-Min
Content	0.09	0.67	0.05	-	-	-
Collaborative KNN	0.08	0.63	0.05	0.29	0.32	0.26
Collaborative MF	0.08	0.64	0.05	0.29	0.32	0.26
Best Match	0.15	0.79	0.076	0.82	0.87	0.77
Focus _{cmp}	0.12	0.73	0.064	0.83	0.88	0.77
Focus _{cl}	0.13	0.74	0.062	0.789	0.84	0.73
Breadth	0.16	0.8	0.076	0.76	0.8	0.72

Table 4: How complete become the goals of the user after s/he follows the recommended actions per list.

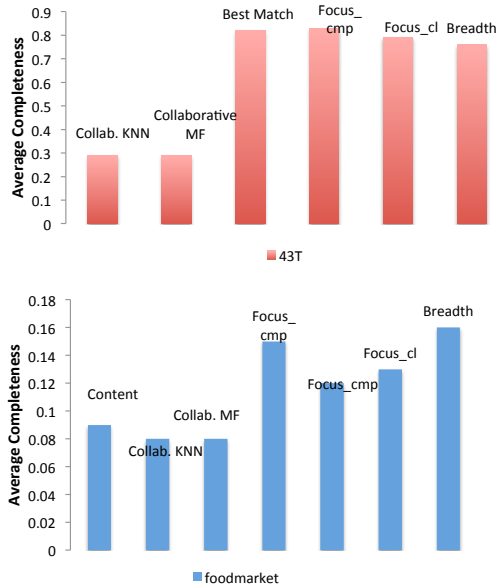


Figure 3: The average goal completeness per list after the user follows the recommended actions in each list.

takes negative values from -1 to 1, with 1 reflecting highly correlated values. Collaborative filtering, which looks into the past actions of similar users for actions that may be of interest to the current user, shows the highest correlation. On the other hand, goal-based methods show negative correlation. They do not promote actions that were popular (frequent) so far. The content-based approach shows a lower correlation than collaborative filtering, which is still high in comparison to the goal-based methods.

C.1.3. Usefulness: the completeness of the goals in the user's goal space after s/he follows the recommended actions. The actions recommended to the user can help her get closer to the fulfillment of (or fulfill) one or more goals. Table 4 shows the average average

(AvgAvg), min (MinAvg) and max (MaxAvg) completeness values for all the recommendation lists formed for the two datasets. Figure 3 shows graphically the average values. These values are estimated by finding first the average, minimum and maximum values of completeness of all the goals that are related to the user considering each list separately. Subsequently, the average for all the recommendation lists is estimated. The goals that we consider in the estimation of goal completeness in the case of the 43T are those that the user has added in the system, while in the case of the food market we consider the whole user's goal space since we do not have any information about which goals the user is pursuing in reality. In the foodmarket dataset, the goal implementation space can be large and not every goal can be fulfilled by performing only 10 actions (i.e., the actions in the recommendation list) in any case. As a consequence, the AvgAvg values in this dataset are not that informative in comparison to those of the 43T dataset.

We observe that *Breadth* and *Best Match* in the first dataset and *Focus_{cmp}* in the second dataset manage the largest completeness (considering both the user activity and the recommended actions), while the lowest contribution is met in the state-of-the-art algorithms. The results are explained by the fact that *Best Match* considers the whole user's goal implementation space, *Breadth* creates a well-connected subspace, while *Focus_{cmp}* selects a single goal (actually a single implementation), if possible, and extends to a few more to complete the recommendation list. *If the user wants to get closer to a wider range of goals, s/he should select Breadth; otherwise (i.e., if s/he is focused on a few goals), s/he should select Focus_{cmp}. Best Match and Focus_{cl} follow.*

C.1.4. Pairwise similarity of the recommended actions (i.e., the corresponding products) in each list. Table 5 shows the pairwise similarity among the retrieved actions in each recommendation list. Due to the lack of widely-accepted domain-specific characteristics for the actions in the 43T dataset, we study the food market

Methods	Pairwise Action Similarity		
	AvgAvg	AvgMax	AvgMin
<i>Content</i>	0.81	1	0.6
<i>Collaborative KNN</i>	0.16	0.5	0.05
<i>Collaborative MF</i>	0.15	0.77	0.04
<i>Best Match</i>	0.33	0.72	0.22
<i>Focus_{cmp}</i>	0.24	0.31	0.21
<i>Focus_{cl}</i>	0.24	0.34	0.19
<i>Breadth</i>	0.33	0.73	0.22

Table 5: Pairwise feature-based similarity among the actions within each recommendation list for the foodmarket dataset.

dataset. AvgAvg is estimated in two steps: first *the average pairwise similarity* considering all the action pairs within each list is estimated, and then the average of the derived values is estimated. The same applies for AvgMax and AvgMin. As expected *Content* shows the highest value with an AvgAvg pairwise value 0.8 and AvgMin value 0.6. Collaborative filtering shows the lowest similarity (AvgAvg 0.15), while all goal-based mechanisms are found in the middle (avg-avg: 0.24-0.33). However, looking at the average maximum pairwise values (AvgMax), we see that the goal-based methods *Best Match* and *Breadth* often share a pair of very similar actions in their lists (on average their max pairwise similarity values are 0.72 and 0.73 respectively). The two *Focus* methods are the goal-based methods that retrieve highly dissimilar actions in most of the cases.

C.1.5. Average percentage of recommended actions that the user has indeed performed (per recommendation list). In the food market dataset, we consider as the user’s current activity a single cart; we have more than one cart for the same user in different time slots though. On the other hand, in the 43T dataset we consider only the 30% of the actions that the users have performed to fulfill their goals. Therefore, we can check whether the different techniques by considering only the actions in the user activity, recommend actions that the user has performed. We should clarify that the average percentage of recommended actions that the user has indeed performed does not reflect the precision of the recommendation tasks since the user has not acted after checking the recommendation lists. In fact, it shows the percentage of the recommended actions for which the user has shown interest at some point. Unlike precision, being able to retrieve actions that the user would anyway perform can be an advantage or a disadvantage for a recommendation technique depending on the view point of the application. If the purpose of the recommendation system is to show to the user unknown actions as well, a very high percentage is not preferable. On the contrary, if the purpose of the system was to provide the user with a discount coupon in order to keep her/him satisfied, a high value would be preferable. Keeping that in mind, we can say that the average percentage represents the Average True Positive Rate. Figure 4 illustrates for each method the Avg TPR for top-5 and top-10 lists. In the top-5 lists, first the *Best Match*, then the *Focus_{cmp}* and *Breadth* show the largest percentage. In the top-10 lists of the foodmarket dataset though, it is the *Content* method that shows the highest percentage. Nevertheless, all the methods show low percentages in the foodmarket dataset. This is explained by the fact that we have no more than 3 carts for each user.

6.1.2 Further Comparison of Goal-based results. Considering the lists derived from the goal-based methods, we have already argued about the fact that the appearance of an action in

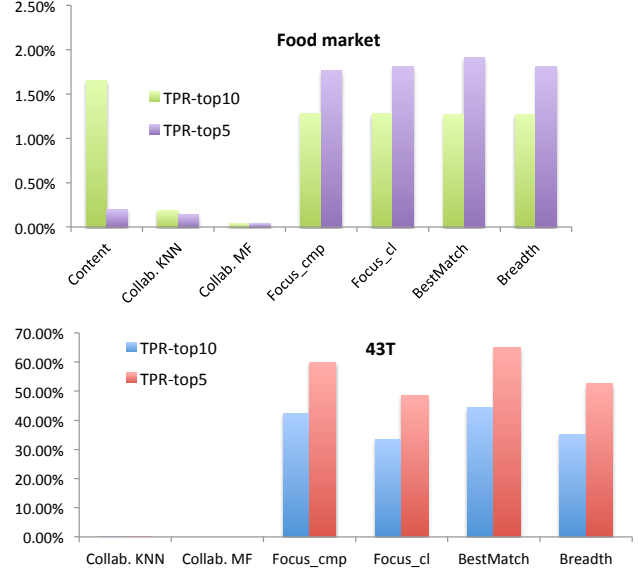


Figure 4: Percentage of recommended actions that the user has indeed performed (True Positive Rate for top-5 and top-10 lists).

the recommendation lists is not correlated to its appearance in the user activities (ref. Table 3). Next we also present whether there exist actions that monopolize the recommendation lists, and how different the recommendation lists formed by the alternative goal-based methods are (*Result Overlapping of Goal-based methods*).

C.2.1. Frequency of Retrieved Items. In recommenders, we do not want certain actions to monopolize the recommendation lists. In the 43T dataset, the frequency of an action in different recommendation lists is very low: *at maximum 0.001*. On the other hand, in the food market dataset, where there are a lot of actions that participate in a great number of implementations (average connectivity 1.2k), the frequency is higher. Figure 5 illustrates that the majority of actions appear with frequency less than 0.2. However, *Best Match* and then *Breadth*, in their effort to serve more than one goal at the same time, repeat the same actions in more recommendation lists (22% and 14% actions respectively with frequency above 0.2). *The actions with high frequency are those that appear frequently in subsets of implementations that share common actions.* Actions that appear in many goal implementations *but together with different actions in each goal implementation* are not selected more frequently. On the contrary, Figure 6 shows that very few actions that appear frequently in the goal implementation sets are in the end selected by any goal-based mechanism. The great majority (more than 92%) of the retrieved actions (by all the goal-based mechanisms) appear in the implementation set with a frequency less than 0.2.

C.2.2. Result Overlapping of Goal-based methods. In Paragraph *C.1.1*, we have presented how different are the results of the goal-based mechanisms from those of the standard recommendation methods, next we present the result overlapping of the goal-based mechanisms. Table 6 illustrates the percentage of common actions in their top-10 lists considering again as input the 21k real carts and the 8k user activities of the food market and the 43T datasets respectively. First of all, we observe a great overlapping in the results of *Best Match* and *Breadth*: 98% and 79% respectively.

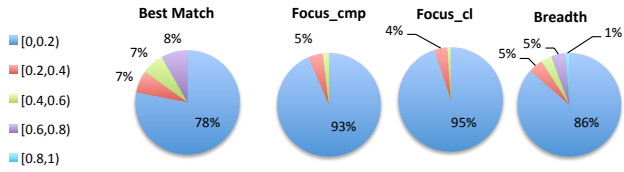


Figure 5: How often the same action appears in the recommendation lists that have been formed for the user activities of the food market dataset. Distribution of actions in frequency ranges.

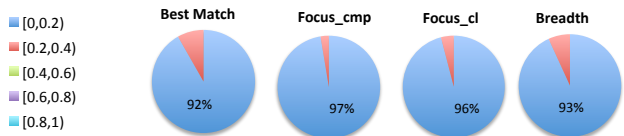


Figure 6: How often the same retrieved action appears in the goal implementation set (herein recipes). Distribution of actions in frequency ranges.

	Food Market	43T
<i>Methods</i>	<i>Overlapping</i>	<i>Overlapping</i>
Best Match- <i>Focus_{cmp}</i>	42%	68%
Best Match-Breadth	98%	79%
<i>Focus_{cmp}</i> -Breadth	44%	71%
<i>Focus_{cl}</i> - <i>Focus_{cmp}</i>	35.6%	78%
<i>Focus_{cl}</i> -Best Match	49%	72%
<i>Focus_{cl}</i> -Breadth	49%	72%

Table 6: Common actions in the top-10 recommendation lists.

The overlapping is higher in the first case because in the food market ingredients participate in a lot of recipes at the same time. Therefore, *Breadth* instead of examining subsets of the user’s goal space to evaluate a certain action, it ends up considering (almost) the whole goal space similarly to *Best match*. In general, the user profile that *Best Match* considers reflects more strongly her/his preference towards a subset of goal(s); thus it (almost) neglects the rest of the goals in the user’s goal space the same way *Breadth* does. Since the two algorithms show similar behavior, *Breadth* is preferred since, as we will see in Subsection 6.2, *Breadth* is significantly more efficient in terms of time.

Moreover, *Focus_{cmp}* and *Focus_{cl}* retrieve the same actions in 35.6% and 78% of the lists respectively. In these cases, there exist goal implementations for which the user has performed most of the actions (completeness) and at the same time these are the implementations with the less remaining actions. Furthermore, *Focus_{cl}* and *Focus_{cmp}* show an overlapping of over 40% and 70% (for the respective datasets) with *Breadth* and *Best Match*. This is justified by the fact that the Focus mechanisms after popping out all the actions of the goal implementation on which they have selected to focus, they move on to another goal implementation. Therefore, they select actions from different goal implementations as *Breadth* and *Best Match* do. Another way to see this is that the latter two algorithms select actions that serve more than one goals at the same time; but that means that the selected actions serve each single goal on its own as well.

Another observation is that the overlapping in the lists for the 43T dataset is larger than in the lists for the food market dataset in all the cases because the action space of the users are wider in

Set	Connectivity	Num Of Distinct Actions	Num Of Implementations
IS	1.2K	380	56K
IS ₂	7.6K	380	282K
IS ₃	15.6K	380	564K
IS ₄	50.3K	380	1.6M
IS ₅	8K	10K	120K
IS ₆	8K	100K	1.2M
IS ₇	8K	1M	12M

Table 7: Goal Implementation Sets.

Alg	IS (56K)	IS ₂ (282K)	IS ₃ (564K)	IS ₄ (1.6M)
	Impls, Conn (1.2K)	Impls, Conn (7.6K)	Impls, Conn (15.6K)	Impls, Conn (50.3K)
Best Match	0.37 s	1.5s	3 s	9.7s
<i>Focus_{cl}</i>	0.001s	0.053	0.096s	0.35s
<i>Focus_{cmp}</i>	0.091s	0.42	0.86s	2.98s
Breadth	0.006s	0.089s	0.089s	0.34s

Table 8: Average Execution Time in implementation sets with high connectivity.

Alg	IS ₅ (10K Actions, 120K Impl)	IS ₆ (100K Actions, 1.2 Impl)	IS ₇ (1M Actions, 12M Impl)
Best Match	0.713s	3.37s	5.38s
<i>Focus_{cl}</i>	0.0017 s	0.0026s	0.0034s
<i>Focus_{cmp}</i>	0.0024s	0.0035s	0.0055s
Breadth	0.0029s	0.0052 s	0.008s

Table 9: Average Execution Time in implementation sets with a large number of actions and implementations.

the latter dataset due to the high action connectivity. Considering a larger set of candidate actions, the algorithms are not forced to select the same actions due to lack of alternatives.

6.2 Scalability

We ran the 4 goal-based strategies (i.e., the 3 strategies plus the extra option for *Focus*) on goal-based association models that have been built based on 7 implementation sets of different characteristics: (a) implementation set size, (b) action set size, and (c) number of implementations in which an action participates on average (*connectivity*). Table 7 illustrates the implementation sets. In sets IS₂, IS₃ and IS₄ the connectivity is stretched up to 50.3K in a set of 1.2M implementations, while in IS₅, IS₆ and IS₇ the size of the action set and the number of implementations increases by 10 times (IS₇ consists of 1M actions and 12M implementations).

Results. Figure 7 illustrates the average time per information need (i.e., per user activity) in secs considering each of the different implementation sets. We observe that the *Best Match* shows the highest execution times in all the cases. The reason is that in goal-based profiles the feature space is not fixed, and thus the representation of the actions is formed on the fly. The rest of the mechanisms show low recommendation time even in the extreme cases of the sets IS₄ and IS₈ (connectivity: 50315, average participation: 19M, and connectivity: 12110, average participation: 137M respectively). On the other hand, *Focus_{cl}* shows the lowest

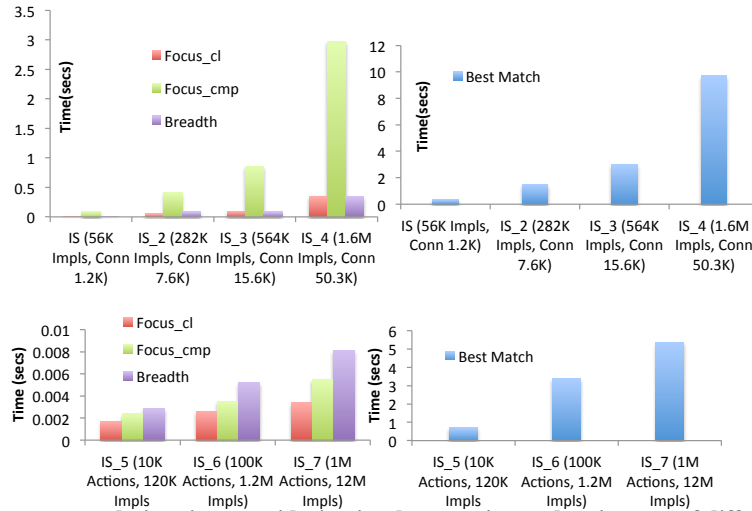


Figure 7: Average recommendation time considering implementation and action sets of different characteristics.

execution time. The difference between $Focus_{cl}$ and $Focus_{cmp}$ results from the two set operations that the mechanisms use, i.e., asymmetric difference and intersection respectively.

In conclusion, the goal-based mechanisms scale well even in sets of millions actions and implementations. Moreover, *the number of actions and implementations alone do not affect much the execution time, it is the higher connectivity that results in higher execution times.*

7 CONCLUSION

Based on the theory that goals rationalize and by consequence trigger user actions, we introduce a family of recommendation approaches that recommend actions seeing them in respect with a number of goals that the users may fulfill through different action sets. We have presented 3 strategies, each one incorporating goals into the scoring of actions in a different way. The action selections of the goal-based mechanisms are not affected by their domain-based similarity with the actions in the user’s activity, nor by the activities of other users. However, they are affected by the benefit of the actions to be recommended to the goals in the user’s goal space. The strategies *Breadth* and *Best Match* focus on more than one goal at a time. In fact, the latter considers all the goals in the goal space independently from the examined action. On the other hand, the *Focus* mechanisms focus on the fulfillment of one goal at a time. Nevertheless, they all increase the average goal completeness in the user’s goal space without retrieving actions that monopolize the goal implementations. Moreover, all the mechanisms create different recommendation lists for different inputs (i.e., user activities). As part of our future work, we have been examining methodologies that enhance the goal-based mechanisms by considering the user preferences on certain domain-specific characteristics, i.e., hybrid goal-based and content-based approaches.

REFERENCES

- [1] Henk Aarts and Ran R. Hassin. 2004. Automatic goal inference and contagion: On pursuing goals one perceives in other people’s behavior. (01 2004), 153–167.
- [2] Marcelo G. Armentano and Analía Amandi. 2009. Goal Recognition with Variable-order Markov Models. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI’09)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1635–1640.
- [3] Suhrid Balakrishnan. 2010. On-demand Set-based Recommendations (*ACM RecSys’10*). ACM, New York, NY, USA, 313–316.

- [4] Richard W. Byrne and Anne E. Russon. 1998. Learning by imitation: A hierarchical approach. *Behavioral and Brain Sciences* 21, 5 (1998), 667–684.
- [5] Sandra Carberry. 2001. Techniques for plan recognition. *User Modeling and User-Adapted Interaction* 11, 1-2 (2001), 31–48.
- [6] Ada S. Chulef, Stephen J. Read, and David A. Walsh. 2001. A Hierarchical Taxonomy of Human Goals. *Motivation&Emotion* 25, 3 (2001), 191–232.
- [7] Mukund Deshpande and George Karypis. 2004. Item-based top-N Recommendation Algorithms. *ACM Trans. Inf. Syst.* 22, 1 (2004), 143–177.
- [8] Yifan Hu, Yehuda Koren, and Chris Volinsky. 2008. Collaborative filtering for implicit feedback datasets. In *IEEE International Conference on Data Mining (ICDM 2008)*. 263–272.
- [9] Neil Hurley and Mi Zhang. 2011. Novelty and Diversity in Top-N Recommendation – Analysis and Evaluation. *ACM Trans. Internet Technol.* 10, 4, Article 14 (March 2011), 30 pages.
- [10] Yuchul Jung, Jihee Ryu, Kyung-min Kim, and Sung-Hyon Myaeng. 2010. Automatic Construction of a Large-scale Situation Ontology by Mining How-to Instructions from the Web. *Web Semant.* 8, 2-3 (July 2010), 110–124.
- [11] S. Keren, A. Gal, and E. Karpas. 2015. Goal Recognition Design for Non-Optimal Agents. In *AAAI*. 3298–3304.
- [12] Jiye Li, Bin Tang, and Nick Cercone. 2004. Applying association rules for interesting recommendations using rule templates. In *Advances in Knowledge Discovery and Data Mining*. Springer, 166–170.
- [13] Dimitra Papadimitriou, Georgia Koutrika, John Mylopoulos, and Yannis Velegrakis. 2016. The Goal Behind the Action: Toward Goal-Aware Systems and Applications. *ACM Trans. Database Syst.* 41, 4, Article 23 (Nov. 2016), 43 pages.
- [14] Paolo Paretì, Ewan Klein, and Adam Barker. 2014. A Semantic Web of Know-how: Linked Data for Community-centric Tasks (*WWW ’14 Companion*). ACM, New York, NY, USA, 1011–1016.
- [15] Donald Patterson, Lin Liao, Dieter Fox, and Henry Kautz. 2003. Inferring High-Level Behavior from Low-Level Sensors. In *Lecture Notes in Computer Science*, Vol. 2864. 73–89.
- [16] Steffen Rendle, Christoph Freudenthaler, and Lars Schmidt-Thieme. 2010. Factorizing Personalized Markov Chains for Next-basket Recommendation (*WWW ’10*). ACM, New York, NY, USA, 811–820.
- [17] Jihee Ryu, Yuchul Jung, Kyung-min Kim, and Sung H. Myaeng. 2010. Automatic Extraction of Human Activity Knowledge from Method-Describing Web Articles. *Proceedings of the 1st Workshop on Automated Knowledge Base Construction* (2010).
- [18] Eldar Sadikov, Jayant Madhavan, Lu Wang, and Alon Halevy. 2010. Clustering Query Refinements by User Intent. In *In (WWW ’10)*. ACM, New York, NY, USA, 841–850.
- [19] J. J. Sandvig, Bamshad Mobasher, and Robin Burke. 2007. Robustness of Collaborative Recommendation Based on Association Rule Mining (*RecSys ’07*). ACM, New York, NY, USA, 105–112.
- [20] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. 2001. Item-based Collaborative Filtering Recommendation Algorithms (*WWW ’01*). ACM, USA, 285–295.
- [21] Markus Strohmaier, Mark Kröll, and Christian Körner. 2009. Automatically annotating textual resources with human intentions. In *Proceedings of the Hypertext 2009*. 355–356.

Very-Low Random Projection Maps

Anastasios Zouzias*
Swiss Re, Switzerland

Michail Vlachos†
IBM Research - Zurich, Switzerland

ABSTRACT

For Big Data analytics, working in low dimensionalities is beneficial for high performance. Instead of projecting onto a single low dimensionality, we examine, both analytically and empirically, the effects on the ‘learning utility’ of the original dataset when combining several very low-dimensional random projections. The embedding proposed exhibits many favorable traits to existing low-dimensional methodologies, such as low runtime and equivalent or better embedding quality.

1 INTRODUCTION

Random linear projections are well studied owing to the Johnson-Lindenstrauss (JL) lemma [11]. The JL lemma states that any n point-set in high-dimensional Euclidean space can be projected into $O(\log n)$ dimensions while accurately preserving all pairwise distances. The lemma is *tight*, in the sense that $\Omega(\log n)$ dimensions are necessary –see [3] and [10] for lower bounds. Although these lower bounds suggest that reducing the dimensionality of the input data further is not feasible while preserving its metric structure, this could be possible by combining the information from *several very low-dimensional* random projections. So here we examine the following: Given any Euclidean set of n points \mathcal{P} in \mathbb{R}^d and a target dimensionality t which is smaller than $O(\log n)$, is it possible to preserve pairwise distances by combining multiple random projections? How many independent random projections would be required? We study this particular question both from a theoretical and a practical perspective. On a theoretical aspect, we derive bounds on the expected number of random projections needed to accurately answer proximity queries (Theorem 3.1). From a practical perspective, we propose an embedding, VLR-Map, that learns the k-Neighborhood structure of the data-points.

2 RELATED WORK

Several approaches exist in the literature that use several random projections for data analysis [2, 5, 8, 12, 13]. We enumerate a representative list of these efforts. The power of several one-dimensional random projections has been exploited by Kleinberg in an algorithm for nearest-neighbor search [12]. Several one-dimensional random projections of the input data are used as proximity tests for a given query point, which is the underlying idea for constructing efficient data-structures for nearest-neighbor search. The use of multiple random projections into an arbitrary small number of dimensions for nearest neighbor search has been also studied by [2], in which the author projects randomly the original data in several independent trials and then

*Work conducted while at IBM Research - Zurich.

†The research leading to these results has received funding from the European Research Council under the European Union’s Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement no 259569.

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

builds a KD-tree data structure for each instance of the projected data. The set of these KD-trees are used for approximately answering nearest-neighbor queries. Designing a family of locality sensitive hashing (LSH) functions shares conceptual similarities to the framework proposed here [8, 13]. LSH principles also use random projections and aggregate the projections. Finally, data embedding techniques also follow a similar methodology. For example, Boostmap [5] learns a data embedding using triplets of inequalities; the learning process is driven by AdaBoost [7]. Boostmap works effectively in practice, but does not offer any formal analytical guarantees on the quality of its embedding, unlike the present work.

3 OUR APPROACH

The intuition behind the algorithm proposed is as follows: Although a *single* very-low-dimensional random projection might not be useful for approximately preserving all pairwise distances of a given set of points, it might be the case that *several* very-low-dimensional random projections are sufficient, when combined appropriately. There are several questions to be addressed: (1) How many dimensions should be chosen when projecting a given dataset? (2) How many different random projections are needed? (3) How should we combine these random projections? In the related literature there do not exist any satisfying answers to the first question, so far. However, we will see shortly, that as Theorem 3.1 suggests, one can set the number of dimensions t to be $\Omega(1/\varepsilon^2)$, where $\varepsilon > 0$ is the desired accuracy. Theorem 3.1 also provides a rigorous answer to the second question. Now, we turn our attention to the third question.

For the sake of presentation, assume that we want to preserve the distance between two input points $\mathbf{p}_1 \in \mathcal{P}$ and $\mathbf{p}_2 \in \mathcal{P}$ with respect to a query point $\mathbf{q} \in \mathbb{R}^d$. The proposed algorithm constructs several independent low-dimensional random projections using random matrices G_1, G_2, \dots, G_l . We view each random projection as a voter that advocates on the proximity of \mathbf{p}_1 (or \mathbf{p}_2) to \mathbf{q} : each random projection votes for (or against) the validity of the predicate $\{\|\mathbf{p}_1 - \mathbf{q}\|_2 < \|\mathbf{p}_2 - \mathbf{q}\|_2\}$ by checking whether the corresponding *projected* points satisfy the above predicate. If $\|\mathbf{p}_1 - \mathbf{q}\|_2 \ll \|\mathbf{p}_2 - \mathbf{q}\|_2$, then it is easy to show that \mathbf{p}_1 will be closer to \mathbf{q} than \mathbf{p}_2 in the projected space with at least constant probability [11]. An unbiased way to combine the votes of all random projections is to take their majority vote. More precisely, if at least half of the random projections vote that \mathbf{p}_1 is closer to \mathbf{q} than \mathbf{p}_2 is, then the algorithm reports that \mathbf{p}_1 is closer to \mathbf{q} , or vice versa.

To complete the description of the above algorithm, we have to specify the required number of independent random projections. Given \mathcal{P} , the following theorem bounds the number of random projections that are needed for the algorithm to be effective with probability $1 - \delta$. The proof of the theorem below (given in the appendix) is based on concentration of measure arguments appropriately combined with ε -net arguments. In more detail, we build a very dense net¹ \mathcal{N} , i.e., (ε/\sqrt{d}) -net, on the unit ball of

¹In a metric space $M = (X, d)$ and $\varepsilon > 0$, an ε -net is a subset \mathcal{N} of X so that for every $x \in X$ there exists $w \in \mathcal{N}$ so that $d(x, w) \leq \varepsilon$.

\mathbb{R}^d and bound the probability that for each pair of points in \mathcal{N} their norms are preserved for the majority of random projections. Although ε -net arguments of this type can be encountered previously in the relevant literature, the next theorem we state is novel, and appears, to the best of our knowledge, for the first time here.

Our main theorem states that if we draw $O(d \log(d) + \log(1/\delta))$ independent random projections then, with probability at least $1 - \delta$, proximity queries between any two points of \mathcal{P} can be answered correctly for well-separated points.

THEOREM 3.1. *Let $A \in \mathbb{R}^{d \times n} = [\mathbf{p}_1, \mathbf{p}_2, \dots, \mathbf{p}_n]$, $0 < \delta < 1$ and $0 < \varepsilon < 1/2$. Fix any integer $t = \Omega(1/\varepsilon^2)$ and let G_1, G_2, \dots, G_l be an i.i.d. sequence of $t \times d$ random-sign matrices rescaled by $1/\sqrt{t}$. If*

$$l \geq \Omega\left(\frac{d \ln(d/\varepsilon^2)}{\varepsilon^2 t} + \ln(1/\delta)\right) \quad (1)$$

then with probability at least $1 - \delta$ the following holds: Let $\mathbf{p}_i, \mathbf{p}_j \in A$ and given any query $\mathbf{q} \in \mathbb{R}^d$ with $(1 + \gamma) \|\mathbf{p}_i - \mathbf{q}\|_2 \leq \|\mathbf{p}_j - \mathbf{q}\|_2$ where $\gamma > 6\varepsilon$ then $\|G_k(\mathbf{p}_i - \mathbf{q})\|_2 < \|G_k(\mathbf{p}_j - \mathbf{q})\|_2$ holds for the majority of indices $k \in [l]$.

REMARK 1. *When we are only interested in answering nearest neighbour queries between points in \mathcal{P} , the parameter d in Equation (1) can be replaced with n . Indeed, repeat the proof of Theorem 3.1 by building an ε -net on the span of \mathcal{P} .*

Theorem 3.1 provides a guarantee on the preservation of nearest neighbor queries and as a direct consequence, it preserves the kNN metric structure of the input dataset. The kNN preservation property implies that the accuracy of the proposed kNN classification method converges to the accuracy of the kNN classifier in the original high dimensionality, as progressively more independent projections are used.

Discussion: Theorem 3.1 suggests that it is possible to bound the distortion even for very-low-dimensional projections. The result may appear pessimistic at first glance because it recommends a prohibitive number, for practical consideration, of $O(d \log(d))$ independent projections. It is important to consider that the analysis is necessarily pessimistic, because it is not based on the characteristics of a particular distribution or structure, but is generic. That is why the bounds may seem large. However, conditioned on the event that we possess a family of projections that satisfy the conclusion of Theorem 3.1, a Chernoff bound implies that only a constant number of projections are required.

LEMMA 3.2. *Sample S indices from $\{1, 2, \dots, l\}$ uniformly at random with replacement. If $S \geq \frac{2(1+2\eta)^2 \ln(1/\theta)}{4\eta^2}$, then with probability at least $1 - \theta$, $0 < \theta < 1$, the sample S will return the same answer as the majority over all $\{G_i\}_{i \in [l]}$.*

PROOF. Let I_i be the indicator random variable corresponding to the success of the i -th sample from S . By hypothesis, $\mathbb{E}[I_i] = 1/2 + \eta$. The multiplicative Chernoff bound implies that $\Pr\left(\sum_{i=1}^S I_i < (1 - \zeta)(1/2 + \eta)S\right) \leq \exp(-\zeta^2 S/2)$ for every $\zeta \geq 0$. Set $\zeta = 1 - \frac{1}{1+2\eta}$ which implies that $(1 - \zeta)(1/2 + \eta)S = S/2$, also $S \geq 2 \ln(1/\theta)/\zeta^2$ implies that $\exp(-\zeta^2 S/2) \leq \theta$. \square

Lemma 3.2 suggests that in practice there is no need to aggregate over all projections, but only over a small number of them. This is verified in the experimental section, in which we demonstrate that in practice substantially fewer number of projections are required for datasets with particular structure (i.e., real-world

datasets). For all our experiments, we set an upper bound of $l = 70$ independent projections which preserved accurately the neighborhood structure. In fact, in the experimental Section 5 one can see that using multiple but lower-dimensional projections is typically better than having a single higher-dimensional projection with the same storage space. The intuition, here, is that introducing randomness is a favorable component in classification, similar, for example, to the approach that random forests also follow.

4 VLR-MAP

Using the previous theoretical results, we now present VLR-Map, standing for Very-Low Random Projection Map. It capitalizes on very-low-dimensional projections which, when combined, yield an effective embedding. The power of the embedding proposed, lies on its simplicity of implementation and low runtime cost. At its core, VLR-Map learns a low-dimensional embedding by drawing independent random projections, until the distances of the kNN neighbors over all points are sufficiently preserved through a voting process in the low-dimensional space.

Training the embedding: Assume a set of $t \times d$ random-projection matrices G_1, G_2, \dots, G_l and an integer $1 \leq k < n$ representing the number of nearest neighbors. For each point $\mathbf{p} \in \mathcal{P}$, we define the i -th nearest neighbor of \mathbf{p} (w.r.t. \mathcal{P}) as $\gamma_i(\mathbf{p})$ for every $1 \leq i \leq k < n$. Define the following, $T(\mathbf{q}, \mathbf{p}_i, \mathbf{p}_j)$ equals 1 if $\|\mathbf{q} - \mathbf{p}_i\|_2 < \|\mathbf{q} - \mathbf{p}_j\|_2$ and -1 , otherwise. Similarly, for every random projection $s = 1, 2, \dots, l$, define

$$\tilde{T}^{(s)}(\mathbf{q}, \mathbf{p}_i, \mathbf{p}_j) := \begin{cases} 1 & \text{if } \|G_s(\mathbf{q} - \mathbf{p}_i)\|_2 < \|G_s(\mathbf{q} - \mathbf{p}_j)\|_2 \\ -1 & \text{otherwise} \end{cases}$$

In essence, $\tilde{T}^{(s)}(\mathbf{q}, \mathbf{p}_i, \mathbf{p}_j)$ gives us the *vote* of the projection matrix G_s regarding the proximity between the vectors \mathbf{q}, \mathbf{p}_i and \mathbf{p}_j . Now, given several different projections/voters, one can define the majority vote over them:

$$\text{Maj}(\mathbf{q}, \mathbf{p}_i, \mathbf{p}_j) := \begin{cases} \mathbf{p}_i & \text{if } \frac{1}{l} \sum_{s=1}^l \tilde{T}^{(s)}(\mathbf{q}, \mathbf{p}_i, \mathbf{p}_j) \geq 0 \\ \mathbf{p}_j & \text{otherwise} \end{cases}$$

Given a query \mathbf{q} , $\text{Maj}(\mathbf{q}, \mathbf{p}_i, \mathbf{p}_j)$ reports which point between \mathbf{p}_i and \mathbf{p}_j is nearest to \mathbf{q} . Following the above discussion, we define the misclassification rate for a given set of random projections given by Eqn. (2).

$$\text{Err}_k(A) = \frac{1}{n \binom{k}{2}} \sum_{\mathbf{p} \in \mathcal{P}} \sum_{i=1}^{k-1} \sum_{j=i+1}^k \mathbf{1}_{\gamma_i(\mathbf{p}) = \text{Maj}(\mathbf{p}, \gamma_i(\mathbf{p}), \gamma_j(\mathbf{p}))}, \quad (2)$$

where $\mathbf{1}_x$ is the indicator function, i.e., $\mathbf{1}_x$ equals to 1 if x is true, and zero otherwise.

The equation measures the average misclassification rate of each point $\mathbf{p} \in \mathcal{P}$ between all pairs of points in the kNN set and will be used as the measure of quality of any embedding.

Given the original high-dimensional points \mathcal{P} , VLR-Map learns the minimum number of random projections that are required to approximately preserve the nearest neighbors using very simple voting principles. We are only interested in preserving the nearest neighbor set of \mathcal{P} , so VLR-Map draws independent random projections until the misclassification rate in the kNN neighborhood is sufficiently small. We measure the error using the distance ordering over all pairs of points of the original kNN neighborhood, i.e., the ordering of the distances between any two nearest neighbor points of \mathbf{p} $\gamma_i(\mathbf{p})$ and $\gamma_j(\mathbf{p})$ for $1 \leq i, j \leq k$. VLR-Map is scalable because its complexity is essentially linear to the dataset size; the algorithm has an $O(nk^2)$ cost per iteration

and the bound of Theorem 3.1 points to the fact that the algorithm will terminate after a finite number of iterations. In the experimental section, we provide further empirical validation on the quality and runtime of our technique and compare it with other embedding methodologies.

Answering kNN queries: After execution of VLR-Map, the resulting set of projected instances of \mathcal{P} can be used for answering proximity queries. Given a query point $\mathbf{q} \in \mathbb{R}^d$ and any two points $\mathbf{p}_i, \mathbf{p}_j \in \mathcal{P}$ we can test whether \mathbf{q} is closer to \mathbf{p}_i or \mathbf{p}_j by using $\text{Maj}(\mathbf{q}, \mathbf{p}_i, \mathbf{p}_j)$. A crucial computational feature when evaluating $\text{Maj}(\cdot, \cdot, \cdot)$ is that it can be effectively approximated by random sampling as previously explained in Lemma 3.2. In practice, a constant number of projections are sufficient for approximately answering proximity queries. So, for any unlabeled query point \mathbf{q} , the algorithm will provide a label based on the consensus voting from all the very-low dimensional classifiers.

5 EXPERIMENTS

We examine the performance and quality of the algorithms presented on several publicly available datasets. [4, 15]. All the datasets are high-dimensional (with dimensionalities varying from 100+ to 10,000), and while they are not very big datasets, they serve well for showcasing the differences in performance and accuracy of the techniques compared. We compare the embedding quality of our approach with traditional random-projection approaches which project on a higher dimensionality that uses the same total space as our methodology. We show that our approach exhibits better classification accuracy and briefly analyze this result.

5.1 Validation of Main Theorem

First, we provide empirical validation for our main result of Theorem 3.1. Recall that the Theorem states essentially that as we increase the number of independent projections, preservation of nearest neighbor structure will be progressively better. Figure 1 plots the embedding error when preserving the 3-NN structure as we progressively increase the number of independent projections. For clarity, we plot the results on four datasets: Email, Gisettd, USPS and MUSK. The results for the other datasets exhibit similar pattern and are omitted.

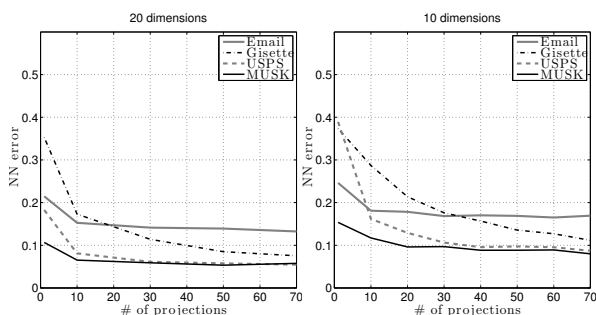


Figure 1: Empirical validation of Theorem 3.1. When projecting to 10 dimensions (right) instead of 20 (left) an equivalent NN-error can be achieved by using additional independent projections.

We use two target dimensions, $t = 20$ and $t = 10$ and average the results over ten independent executions. Observe that Figure 1 validates the main result, because by using a lower target dimensionality of $t = 10$ an equivalent error of the higher target

dimensionality $t = 20$ can still be achieved through the use of additional projections.

Therefore, the power of the methodology proposed lies in its simplicity; by using and combining additional very-low-dimensional projections we can substantially influence the quality of the embedding and of the distance preservation. It is important to underscore that because the individual projections are independent of each other, the classifiers operating on each of the projections are totally segregated and could be run in parallel. Because each classification is independent of the others (aside from the small voting phase), given sufficient CPUs/cores, the overall runtime of our approach should remain approximately constant, even under increasing cardinality of projections/classifiers.

5.2 Random Projection Methodologies

We compare the classification error of various methodologies based on random projections: VLR-Map, a kNN classifier using a *single* projection onto t dimensions (sRP_t), a kNN classifier with a *single* projection onto $t \cdot l$ dimensions ($\text{sRP}_{t \cdot l}$) and locality-sensitive-hashing (LSH) [8]. $\text{sRP}_{t \cdot l}$ is included in the comparisons to compare the performance of VLR-Map with the traditional single random projection methodology which *uses the same space* ($\text{sRP}_{t \cdot l}$ has the same number of coordinates with our approach). The comparison with LSH is also done under fair settings; the number of hash functions equals the number of projections l of VLR-Map, and the number of bins for each hash function equals the projected dimensionality t . Finally, for reference, we also include the classification accuracy of the kNN classifier on the original high-dimensional points (kNN). We report the results for $k = 3$ nearest neighbors and target dimensionality of $t = 30$ in Table 1. The experiments indicate that our approach can, in fact, achieve comparable (or sometimes even better) performance than the kNN classifier which operates in the original data dimensionality. An advantage of our framework is that it is computationally lighter than a traditional kNN classifier, because in very low dimensional spaces (in which our framework operates) the nearest-neighbor search can be executed efficiently using data structures, such as KD-trees. However, in higher-dimensional spaces, the performance of these techniques degrades rapidly as validated both analytically and empirically in many studies [9].

More importantly, the results suggest that our approach outperforms the traditional projection methodology which uses the same space (i.e., a single random projection at dimensionality $t \cdot l$). One can think of this as quite analogous to the concept behind random forests [6]. Having multiple random classifiers (unweighted in our case) can boost classification and also introduce robustness. Finally, in Table 2 we report the runtime for one experiment on three datasets for the various techniques based on random projections. VLR-Map and $\text{sRP}_{t \cdot l}$ have equivalent runtime, while LSH is costlier.

6 CONCLUSION

Our main theorem highlights that it is feasible to combine many very-low-dimensional projections and guarantee a bounded distortion on the original distances. From a practical viewpoint, the embedding proposed, VLR-Map, exhibits many favorable traits, such as: i) simplicity of implementation, and, ii) scalability: significantly reduced run-time compared to state-of-art embedding techniques with comparable accuracy.

Classification Error on test data (%)														
Method	3-NN	SRP _t	VLR-Map	sRP _{t,l}	LSH	VLR-Map	sRP _{t,l}	LSH	VLR-Map	sRP _{t,l}	LSH	VLR-Map	sRP _{t,l}	LSH
no. projections $l =$	10					30			50			70		
MNIST	2.95	9.25	4.29	3.23	10.14	3.73	3.11	3.77	3.61	3.12	4.64	3.5	3.15	8.95
COIL	1.15	2.3	0.92	1.23	0.69	0.85	1.15	1.85	0.38	1.15	5.0	0.77	1.23	11.47
Email	18	19.84	13.71	15.76	26.73	12.91	16.04	26.78	12.38	16.38	26.69	12.27	16.6	26.69
lights	1.9	6.06	2.41	2.49	3.96	2.02	2.33	3.96	2.33	2.25	3.96	2.3	2.33	3.81
PIE	11.7	22.5	12.2	12.02	16.49	10.1	11.51	18.36	9.8	11.98	19.76	9.49	11.79	22.21
USPS	5.7	11.5	6.53	5.7	6.77	5.23	5.47	12.5	4.97	5.53	45.33	4.67	5.87	71.53
GISETTE	3.0	25.1	13.17	6.07	2.6	7.8	3.9	2.6	6.7	3.6	2.6	6.07	3.3	2.6
MUSK	4.21	9.33	4.98	5.82	9.47	4.91	5.89	10.25	4.98	5.47	12.00	4.42	5.12	16.07

Table 1: VLR-Map offers overall better accuracy (on t dimensions using l projections) than LSH or a single projection that uses that same space (sRP_{t,l}). Red color denotes better values (smaller classification error).

	sRP _{t,l}	VLR-Map	LSH
MNIST	0.87	1.22	1.95
USPS	0.06	0.10	0.85
MUSK	0.01	0.04	0.34

Table 2: Time comparison (in sec) between a single random projection (sRP_{t,l}), VLR-Map ($l = 30$ and $t = 10$) and LSH. VLR-Map and single random projection exhibit equivalent runtime, while LSH computations are more expensive.

REFERENCES

- [1] D. Achlioptas. Database-friendly Random Projections: Johnson-Lindenstrauss with Binary Coins. *J. Comput. Syst. Sci.*, 66(4):671–687, 2003.
- [2] Y. S. Ahmed. Multiple Random Projections for Fast, Approximate Nearest Neighbor Search in High Dimensions.
- [3] N. Alon. Problems and results in extremal combinatorics, part i. *Discrete Math.*, 273, 2003.
- [4] A. Asuncion and D. J. Newman. UCI machine learning repository, 2007.
- [5] V. Athitsos, J. Alon, S. Sclaroff, and G. Kollios. Boostmap: An embedding method for efficient nearest neighbor retrieval. *IEEE Trans. Pattern Anal. Mach. Intell.*, 30:89–104, 2008.
- [6] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.
- [7] Y. Freund and R. E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *J. Comput. Syst. Sci.*, 55(1):119–139, Aug. 1997.
- [8] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. *In Proc. VLDB*, pages 518–529, 1999.
- [9] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. *In Proc. STOC*, pages 604–613, 1998.
- [10] T. S. Jayram and D. P. Woodruff. Optimal Bounds for Johnson-Lindenstrauss Transforms and Streaming Problems with Sub-Constant Error. *In Proc. SODA*, pages 1–10, 2011.
- [11] W. B. Johnson and J. Lindenstrauss. Extensions of Lipschitz mappings into a Hilbert space. *Contemporary mathematics*, 26(189-206):1–1, 1984.
- [12] J. M. Kleinberg. Two Algorithms for Nearest-Neighbor Search in High Dimensions. *In Proc. STOC*, pages 599–608. ACM, 1997.
- [13] E. Kushilevitz, R. Ostrovsky, and Y. Rabani. Efficient Search for Approximate Nearest Neighbor in High Dimensional Spaces. *In Proc. STOC*, pages 614–623, 1998.
- [14] J. Matousek. *Lectures on Discrete Geometry*. Springer, 2002.
- [15] K. P. Murphy. A collection of MATLAB data sets used by PMTK. <https://github.com/probml/pmtkdata>, 2014.

PROOF. of Theorem 3.1: Assume two points \mathbf{x}, \mathbf{y} of the pointset A and a query point $\mathbf{q} \in \mathbb{R}^d$. The goal is to decide whether \mathbf{q} is nearest to \mathbf{x} or to \mathbf{y} . Without loss of generality, we can assume that \mathbf{q} is the origin by linearity (otherwise apply the argument below to the vectors $\mathbf{x} - \mathbf{q}$ and $\mathbf{y} - \mathbf{q}$). So, it suffices to argue that one can check the distances of \mathbf{x} and \mathbf{y} .

Let G be a $t \times d$ random-sign matrix rescaled by $1/\sqrt{t}$, i.e., a matrix whose entries are i.i.d. uniformly distributed r.v. on $\{\pm 1\}$. For any $0 < \epsilon < 1/2$ and $\mathbf{z} \in \mathbb{R}^d$, the following bound is well-known [1]

$$\mathbb{P}\left(\left|\frac{\|G\mathbf{z}\|_2^2}{\|\mathbf{z}\|_2^2} - 1\right| > \epsilon\right) \leq \exp(-C_{\text{JL}}\epsilon^2 t) \quad (3)$$

where $C_{\text{JL}} > 0$ is an absolute constant. More precisely, C_{JL} is the constant of the Johnson-Lindenstrauss lemma with random sign matrices. For any $\mathbf{z}_1, \mathbf{z}_2 \in \mathcal{S}^{d-1}$, define the following event

$$\mathcal{E}_j(\mathbf{z}_1, \mathbf{z}_2) := \left\{ \left| \|G_j \mathbf{z}_1\|_2^2 - 1 \right| < \epsilon \text{ and } \left| \|G_j \mathbf{z}_2\|_2^2 - 1 \right| < \epsilon \right\}.$$

It follows by (3) that for every $j = 1, \dots, l$: $\mathbb{P}(\mathcal{E}_j^c(\mathbf{z}_1, \mathbf{z}_2)) \leq 2\exp(-C_{\text{JL}}\epsilon^2 t)$. Moreover, let us define the event $\text{Maj}(\mathbf{z}_1, \mathbf{z}_2) = \{\mathcal{E}_j(\mathbf{z}_1, \mathbf{z}_2) \text{ holds for } \geq \lceil l/2 \rceil \text{ indices } j.\}$. Let us first bound the probability

$$\begin{aligned} \mathbb{P}(\text{Maj}(\mathbf{z}_1, \mathbf{z}_2)^c) &= \sum_{s=\lceil l/2 \rceil}^l \mathbb{P}(\mathcal{E}_j(\mathbf{z}_1, \mathbf{z}_2)^c \text{ for exactly } s \text{ indices}) \\ &\leq \sum_{s=\lceil l/2 \rceil}^l \binom{l}{s} \mathbb{P}(\mathcal{E}_1(\mathbf{z}_1, \mathbf{z}_2)^c)^s \leq \sum_{s=\lceil l/2 \rceil}^l \binom{l}{s} 2^s \exp(-C_{\text{JL}}\epsilon^2 s t) \\ &\leq \sum_{s=\lceil l/2 \rceil}^l \binom{l}{s} 2^l \exp(-C_{\text{JL}}\epsilon^2 \lceil l/2 \rceil t) \\ &\leq 2^l \exp(-C_{\text{JL}}\epsilon^2 \lceil l/2 \rceil t) \sum_{s=0}^l \binom{l}{s} \leq 2^{2l} \exp(-C_{\text{JL}}\epsilon^2 t l/2). \end{aligned}$$

Now, we bound the probability that the majority of random projections $\{G_j\}$ preserves the norms of every pair of points in \mathcal{N} , where \mathcal{N} is an (ϵ/\sqrt{d}) -net of \mathcal{S}^{d-1} . Recall that $|\mathcal{N}| \leq (3\sqrt{d}/\epsilon)^d$ [14]. Namely, we bound the following $\mathbb{P}(\forall \mathbf{z}_1 \in \mathcal{N}, \forall \mathbf{z}_2 \in \mathcal{N} : \text{Maj}(\mathbf{z}_1, \mathbf{z}_2) = 1 - \mathbb{P}(\exists \mathbf{z}_1, \mathbf{z}_2 \in \mathcal{N}, \text{Maj}(\mathbf{z}_1, \mathbf{z}_2)^c))$. The last quantity can be bounded as follows:

$$\begin{aligned} \mathbb{P}(\exists \mathbf{z}_1, \mathbf{z}_2 \in \mathcal{N}, \text{Maj}(\mathbf{z}_1, \mathbf{z}_2)^c) &\leq \sum_{\mathbf{z}_1, \mathbf{z}_2 \in \mathcal{N}} \mathbb{P}(\text{Maj}(\mathbf{z}_1, \mathbf{z}_2)^c) \\ &\leq |\mathcal{N}|^2 2^{2l} \exp(-C_{\text{JL}}\epsilon^2 t l/2) \leq \left(\frac{\sqrt{d}}{3\epsilon}\right)^{2d} 2^{2l} \exp(-C_{\text{JL}}\epsilon^2 t l/2) \\ &= \exp(-l(C_{\text{JL}}\epsilon^2 t/2 - \ln(4)) + 3d \ln(d/\epsilon^2)), \end{aligned}$$

where in the first inequality we used the union bound and the final inequality by (3). Assume that $\epsilon^2 t > 2 \ln(4)/C_{\text{JL}}$ ($t = \Omega(1/\epsilon^2)$ by assumption), hence if $l \geq \frac{4d \ln(d/\epsilon^2)}{C_{\text{JL}}\epsilon^2 t - 2 \ln(4)} + \ln(1/\delta)$ then

$$\mathbb{P}(\forall \mathbf{z}_1 \in \mathcal{N}, \forall \mathbf{z}_2 \in \mathcal{N} : \text{Maj}(\mathbf{z}_1, \mathbf{z}_2) \geq 1 - \delta).$$

From now on, assume that the following event holds

$$\{\forall \mathbf{z}_1 \in \mathcal{N}, \forall \mathbf{z}_2 \in \mathcal{N} : \text{Maj}(\mathbf{z}_1, \mathbf{z}_2)^c\}. \quad (4)$$

Next we prove that (assuming (4)) if $(1 + \gamma) \|\mathbf{x}\|_2 \leq \|\mathbf{y}\|_2$, then the majority of the fixed projections $\{G_i\}_{i \in [l]}$ will satisfy $\|G_i \mathbf{x}\|_2 < \|G_i \mathbf{y}\|_2$.

Indeed, let \mathbf{x}_N and \mathbf{y}_N be the net points that are nearest to $\mathbf{x}/\|\mathbf{x}\|_2$ and $\mathbf{y}/\|\mathbf{y}\|_2$, respectively. Namely, it holds that $\|\mathbf{x}/\|\mathbf{x}\|_2 - \mathbf{x}_N\|_2 \leq \epsilon/\sqrt{d}$ and $\|\mathbf{y}/\|\mathbf{y}\|_2 - \mathbf{y}_N\|_2 \leq \epsilon/\sqrt{d}$. Conditioning on the event in Eq. (4) implies that for the majority of $\{G_j\}_{j \in [l]}$

$$\begin{aligned} \|G_j \mathbf{x}\|_2 &= \|\mathbf{x}\|_2 \|G_j \mathbf{x}/\|\mathbf{x}\|_2 - G_j \mathbf{x}_N + G_j \mathbf{x}_N\|_2 \\ &\leq \|\mathbf{x}\|_2 (\|G_j\|_2 \|\mathbf{x}/\|\mathbf{x}\|_2 - \mathbf{x}_N\|_2 + \|G_j \mathbf{x}_N\|_2) \\ &\leq \|\mathbf{x}\|_2 (\|G_j\|_2 \epsilon/\sqrt{d} + 1 + \epsilon) \leq (1 + 2\epsilon) \|\mathbf{x}\|_2, \end{aligned}$$

where the first inequality is triangle inequality combined with standard matrix norms, the second inequality follows by Eq. (4) and the definition of \mathbf{x}_N and the last inequality follows since $\|G_j\|_2 \leq \|G_j\|_F \leq \sqrt{d}$.

Similarly, for the majority of the indices j ,

$$\begin{aligned} \|G_j \mathbf{y}\|_2 &= \|\mathbf{y}\|_2 \|G_j \mathbf{y}/\|\mathbf{y}\|_2 - G_j \mathbf{y}_N + G_j \mathbf{y}_N\|_2 \\ &\geq \|\mathbf{y}\|_2 (\|G_j \mathbf{y}_N\|_2 - \|G_j\|_2 \|\mathbf{y}/\|\mathbf{y}\|_2 - \mathbf{y}_N\|_2) \\ &\geq \|\mathbf{y}\|_2 (1 - \epsilon - \|G_j\|_2 \epsilon/\sqrt{d}) \geq (1 - 2\epsilon) \|\mathbf{y}\|_2. \end{aligned}$$

Therefore, we conclude that the ratio $\|G_j \mathbf{y}\|_2 / \|G_j \mathbf{x}\|_2$ is at least $\frac{(1-2\epsilon)\|\mathbf{y}\|_2}{(1+2\epsilon)\|\mathbf{x}\|_2} \geq \frac{(1-2\epsilon)}{(1+2\epsilon)}(1 + \gamma) > 1$ for the majority of random projections $\{G_j\}_{j \in [l]}$ as $\gamma > 6\epsilon$. \square

Interval Count Semi-Joins

Panagiotis Bouros

Institute of Computer Science
 Johannes Gutenberg University Mainz, Germany
 bouros@uni-mainz.de

Nikos Mamoulis

Dept. of Computer Science & Engineering
 University of Ioannina, Greece
 nikos@cs.uoi.gr

ABSTRACT

Interval joins find applications in several domains, including temporal and spatial databases, uncertain data management, streaming data processing. In this paper, we study the evaluation of an interval count semi-join (*ICSJ*) operation that can be used for selecting or ranking intervals based on the number of join pairs they appear in. We extend the state-of-the-art algorithm for interval joins to evaluate *ICSJ* at the cost of only scanning the sorted interval endpoints.

1 INTRODUCTION

The interval join (*IJ*) is an important and well-studied operation that finds several applications. In its most widely used definition, the interval join takes as input two collections of intervals R and S , and outputs the pairs $(r, s) \in R \times S$, such that intervals r and s overlap¹. In temporal databases [8], tuples are associated with validity intervals and interval joins can be used to find pairs of tuples with overlapping validity (e.g., find pairs of employees who worked in two enterprises during overlapping time periods). In spatial databases, multidimensional overlap joins reduce to interval joins if the spatial extent of the objects is represented by a set of intervals with the help of space-filling curves [11]. In probabilistic databases, values in continuous domains are often represented by intervals of values which have non-zero probability [4]. Finally, in applications that process streaming data, values read from different streams can be joined by (often parameterized) temporal windows [7]. Such sliding window joins can be modeled as overlap joins, if the values are extended by the window lengths and modeled as intervals. A number of single-processor [2, 5, 6, 13, 14] and parallel [1, 3, 9] algorithms for interval joins have been proposed. Among them, methods that are based on plane-sweep prevail due to their optimal worst-case complexity and their efficient implementations [1, 13].

In this paper, we study the efficient evaluation of an *Interval Count Semi-Join (ICSJ)* operation, where the objective is to find how many pairs in the interval join $IJ(R, S)$ result include each $r \in R$. For example, consider interval collections $R = \{r_1, r_2, r_3\}$ and $S = \{s_1, s_2, s_3, s_4\}$ depicted in Figure 1. $ICSJ(R, S) = \{(r_1, 2), (r_2, 1), (r_3, 3)\}$ because r_1, r_2 and r_3 overlap with 2, 1, and 3 intervals from S , respectively. *ICSJ*(R, S) can be seen as a case of temporal aggregation on S , using R as the set of fixed intervals [12]. The result of *ICSJ* can be used to select or rank objects that are associated with the intervals in R based on the number of intervals in S they intersect. For example, if R includes the employment periods of employees in company A and S includes the periods of employees in company B, we may wish to find the k employees in A whose employment time overlaps with that of

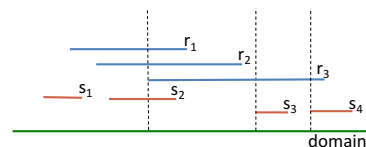


Figure 1: Collections $R = \{r_1, r_2, r_3\}$ and $S = \{s_1, s_2, s_3, s_4\}$.

the most employees in B (e.g., employee r_3 in Figure 1, if $k = 1$). This problem can be solved by ranking the *ICSJ* output, or by using a priority queue to keep track of the top- k *ICSJ* results while they are computed.

To our knowledge, *ICSJ* has not been adequately studied to date. Top- k count semi-joins have been studied in relational [15] and spatial databases [16], but the techniques in these studies are unsuited for *ICSJ*, as they apply on different data domains and they make use of indices. We present an efficient *smart counting* algorithm for evaluating *ICSJ*, which applies on two sorted input collections R and S and extends the state-of-the-art sweeping based *IJ* algorithm. The algorithm bears only the minimal cost of scanning the sorted inputs. Experiments on four real datasets show that it is orders of magnitude faster than simpler alternatives.

2 BACKGROUND

Related work on *IJ* includes techniques based on indexing or partitioning [2, 5, 6] and methods that sort inputs R, S to perform merge-join [14] or plane-sweep based join [1, 13]. Recent studies [1, 13] focused on in-memory processing and showed that plane-sweep based techniques are superior to other methods.

Algorithm 1 describes a plane-sweep based algorithm for interval joins. Initially, the domain points (or simply points) of all intervals in R, S are extracted and sorted into list L (Lines 2–3). Intuitively, L defines the stops of an imaginary line that sweeps the domain; so, the sweep line stops both at the start and the end point of an interval. An *active set* A^R and A^S is initialized for each of the two input collections. Sets A^R and A^S keep track of the intervals that are currently “open” (i.e., their start point has been encountered but not their end point). Each point in list L is accessed in order; if it is a start point of an interval, e.g., $r \in R$, r is guaranteed to overlap all intervals in A^S . Therefore, all pairs in $\{r\} \times A^S$ are reported. For example, consider Figure 1 and assume that the start point of r_3 is currently accessed (i.e., the sweep line is the leftmost vertical line). The active set of S is $A^S = \{s_2\}$, hence the algorithm outputs pair (r_3, s_2) as part of the join result. If an end point is encountered by the sweep line (Lines 11–12 and 18–19 of Algorithm 1), the corresponding interval is no longer open, so it is removed from its respective active set.

Assuming an efficient data structure, which performs insertions and deletions to the active sets in constant time (e.g., a hash table), Algorithm 1 computes the join in $O(|R| + |S| + K)$ time, where K is the number of result pairs, excluding the sorting cost of list L . Note that when a start point is encountered, an active set should be scanned to generate join results. Scanning a hash table

¹Two intervals overlap (or intersect) if they share at least one common value.

ALGORITHM 1: Plane-Sweep based Interval Join

Input : collections of intervals R and S
Output : set of all intersecting interval pairs $(r, s) \in R \times S$
Variables : interval points list L , active sets A^R, A^S

```
1  $A^R \leftarrow \emptyset, A^S \leftarrow \emptyset$ ;  $\triangleright$  sets of active intervals from  $R$  and  $S$ 
2  $L \leftarrow$  start and end points of all intervals in  $R \cup S$ ;
3 sort  $L$ ;
4 while  $L$  is not depleted do
5    $p \leftarrow$  next point in  $L$ ;
6   if  $p$  originates from collection  $R$  then
7      $r \leftarrow$  interval in  $R$  where  $p$  belongs;
8     if  $p$  is a start point then
9       add  $r$  to  $A^R$ ;  $\triangleright r$  is open
10      output  $\{(r, s) : \forall s \in A^S\}$ ;  $\triangleright r$  overlaps all
11      intervals in  $A^S$ 
12      else remove  $r$  from  $A^R$ ;  $\triangleright r$  no longer open
13    else
14       $s \leftarrow$  interval in  $S$  where  $p$  belongs;
15      if  $p$  is a start point then
16        add  $s$  to  $A^S$ ;  $\triangleright s$  is open
17        output  $\{(r, s) : \forall r \in A^R\}$ ;  $\triangleright s$  overlaps all
18        intervals in  $A^R$ 
19      else remove  $s$  from  $A^S$ ;  $\triangleright s$  no longer open
```

is expensive as it incurs random accesses in memory; to this end, Piatov et al. [13] designed the *gapless hash map* which efficiently supports all three insert, remove and getNext operations.² Finally, in [1], an implementation of the plane-sweep based interval join that replaces active sets (e.g., A^R) by forward scans to the other collection (i.e., S), was investigated and optimized.

3 EVALUATING ICSJ

We now investigate how the plane-sweep based Algorithm 1 can be extended to efficiently evaluate interval count semi-joins. Recall that the goal is to count for every interval $r \in R$, the number of overlapping intervals from S .

A naive approach for computing $ICSJ(R, S)$ is to evaluate $IJ(R, S)$ first. Then in an aggregation step, we need to sort or hash the (r, s) join pairs by their first element, and count and report the number of pairs for each $r \in R$. Naturally, this method is at least as expensive as the interval join problem. In fact, since the number of overlapping intervals can be much greater than the sizes $|R|$ and $|S|$ of the two inputs, the cost of sorting or hashing the $IJ(R, S)$ results may dominate the overall evaluation cost.³

To address these shortcomings, we next present two methods, which extend Algorithm 1 to directly compute $ICSJ(R, S)$.

3.1 The Simple Counting Approach

We first discuss an intuitive extension to Algorithm 1 based on the following observation. When the start point of an interval (e.g., $r \in R$) is encountered, the algorithm scans the active set of the other collection (i.e., A^S) to produce join pairs $\{r\} \times A^S$. As our objective is only to count the intervals from S that overlap interval r , we do not have to scan active set A^S ; instead, we only need to add its size $|A^S|$ to a dedicated counter for r . Note that

²In [13], Algorithm 1 is presented as the *Endpoint-Based Interval* (EBI) Join algorithm.

³Note also that this naive method is not suitable for in-memory evaluation of count semi-joins due to buffering all $IJ(R, S)$ result pairs.

ALGORITHM 2: Simple Counting

Input : collections of intervals R and S
Output : for each $r \in R$, $|s : s \in S, \text{ and } r \text{ intersects } s|$
Variables : interval points list L , active sets A^R, A^S , hash table C

```
1  $A^R \leftarrow \emptyset, A^S \leftarrow \emptyset$ ;  $\triangleright$  sets of active intervals from  $R$  and  $S$ 
2  $L \leftarrow$  start and end points of all intervals in  $R \cup S$ ;
3 sort  $L$ ;
4 while  $L$  is not depleted do
5    $p \leftarrow$  next point in  $L$ ;
6   if  $p$  originates from collection  $R$  then
7      $r \leftarrow$  interval in  $R$  where  $p$  belongs;
8     if  $p$  is a start point then
9       add  $r$  to  $A^R$ ;  $\triangleright r$  is open
10       $C[r] \leftarrow |A^S|$ ;  $\triangleright$  initialize counter for  $r$ 
11      else
12        remove  $r$  from  $A^R$ ;  $\triangleright r$  no longer open
13        output  $(r, C[r])$ ;
14        delete  $C[r]$ ;
15      else
16         $s \leftarrow$  interval in  $S$  where  $p$  belongs;
17        if  $p$  is a start point then
18          add  $s$  to  $A^S$ ;  $\triangleright s$  is open
19          foreach  $r \in A^R$  do
20             $C[r] \leftarrow C[r] + 1$ ;  $\triangleright s$  overlaps all
21            intervals in  $A^R$ 
22          else remove  $s$  from  $A^S$ ;  $\triangleright s$  no longer open
```

this is not the final value of this counter, because r may also overlap with intervals from S that start later. Nevertheless, we can eliminate the overhead of scanning the active set of collection S , which in practice also means that a typical hash table can be used for A^S instead of an optimized special structure (e.g., the gapless hash map of [13]) as we only need to support efficient insertions and deletions. In contrast, we still have to support efficient scans for active set A^R , because for each encountered start point from S , we have to scan A^R in order to increase the counters of all open intervals from r .

Algorithm 2 is a pseudocode of this *Simple Counting* approach. Compared to Algorithm 1, we define a hash table C to maintain the dedicated counter for each open interval from collection R . Further, as already discussed, Simple Counting initializes counter $C[r]$ in Line 10, when the start point of an interval $r \in R$ is seen. The counter for each $r \in A^R$ is then increased by 1, when the start of an interval from S is seen in Lines 19–20. Finally, as soon as the end point of r is accessed, counter $C[r]$ is finalized and hence removed from hash table C and reported as result (Lines 13–14). Consider for example r_3 in Figure 1; when accessing the start point of the interval, counter $C[r_3]$ is initialized to 1 as $A^S = \{s_2\}$. After the next two stops of the sweep line marked in the figure, i.e., when the start points of intervals s_2 and s_3 are encountered, $C[r_3]$ is increased to 3. Algorithm 2 is similar to the general approach for temporal aggregation, proposed in [12].

Simple Counting is expected to always outperform the naive solution; recall that the latter needs to completely evaluate IJ as its first step. On the other hand, Algorithm’s 2 cost is in same order to Algorithm 1 as half of the IJ results are still computed, i.e., the pairs generated in Lines 19–20 when encountering start points from S . Note that Simple Counting is also charged with the book-keeping cost for the counters of hash table C . In view

ALGORITHM 3: Smart Counting

Input : collections of intervals R and S
Output : for each $r \in R$, $|s : s \in S, \text{ and } r \text{ intersects } s|$
Variables : interval points list L , hash table C

```
1  $|A^S| \leftarrow 0$ ;  $\triangleright$  active set counter for intervals from  $S$ 
2  $g \leftarrow 0$ ;  $\triangleright$  global counter
3  $L \leftarrow$  start and end points of all intervals in  $R \cup S$ ;
4 sort  $L$ ;
5 while  $L$  is not depleted do
6    $p \leftarrow$  next point in  $L$ ;
7   if  $p$  originates from collection  $R$  then
8      $r \leftarrow$  interval in  $R$  where  $p$  belongs;
9     if  $p$  is a start point then
10       $C[r] \leftarrow |A^S| - g$ ;  $\triangleright$  initialize counter for  $r$ 
11     else
12       $C[r] \leftarrow C[r] + g$ ;
13      output  $(r, C[r])$ ;
14      delete  $C[r]$ ;
15     else
16       $s \leftarrow$  interval in  $S$  where  $p$  belongs;
17      if  $p$  is a start point then
18         $|A^S| \leftarrow |A^S| + 1$ ;  $\triangleright$  increase active set counter
19         $g \leftarrow g + 1$ ;  $\triangleright$  increase global counter
20      else
21         $|A^S| \leftarrow |A^S| - 1$ ;  $\triangleright$  decrease active set counter
```

of these shortcomings, we next present a significantly faster extension to Algorithm 1.

3.2 The Smart Counting Approach

The main idea behind the *Smart Counting* extension to Algorithm 1 is to maintain cheap statistics about the intervals from S instead of keeping track of A^S at every position of the sweep line. Algorithm 3 is the pseudocode of the Smart Counting approach. We now discuss its key features.

First, we observe that only the size of active set A^S is in fact needed for the *ICSJ* computation. Although the Simple Counting algorithm presented in Section 3.1 keeps track of the open intervals from S , the contents of A^S are never scanned and only $|A^S|$ is used on Line 10 of Algorithm 2. Hence, we can replace the hash table of active set A^S by a simple size counter $|A^S|$; when the start point of an interval $s \in S$ is encountered, this counter is increased by 1 (Line 18) while after an end point from S is accessed the same counter is reduced by 1 (Line 21). Next, we define a *global counter* g to keep track of the *number* of intervals from S that have opened (regardless whether their end point is already accessed or not). Similar to $|A^S|$, counter g is increased by 1 in Line 19 when a start point from collection S is seen but never decreased which means that $g \geq |A^S|$ always holds.

By combining counters $|A^S|$ and g , we are able to compute the number of intervals from S that opened or were open in-between the start and the end point of an interval $r \in R$. In specific, we initialize the dedicated counter $C[r] = |A^S|$ when the start point of r is encountered but then subtract the value of global counter g (Line 8). Compared to Algorithm 2, notice that we no longer maintain open intervals from R to active set A^R ; instead we employ hash table C to store the current value of r 's dedicated counter. After the end point of interval r is seen, we just need to add back the current value of g to $C[r]$ and report

Table 1: Characteristics of experimental datasets

	FLIGHTS	BOOKS	GREEND	WEBKIT
Cardinality	445,827	2,312,602	110,115,441	2,347,346
Domain duration (secs)	2,750,280	31,507,200	283,356,410	461,829,284
Shortest interval (secs)	1,261	1	1	1
Longest interval (secs)	42,301	31,406,400	59,468,008	461,815,512
Avg. interval duration (secs)	8,791	2,201,320	16	33,206,300
Distinct domain points	41,975	5,330	182,028,123	174,471

result $(r, C[r])$ (Lines 12–13). This procedure guarantees that we will end up with the correct value of $C[r]$, because the difference from global counter g corresponds to the number of intervals from S that opened after r 's start point. Note that these intervals overlap with r but were not considered when $C[r]$ was initialized.

We expect the Smart Counting approach to significantly outperform Simple Counting as the cost of maintaining and scanning active sets A^R, A^S is completely eliminated. Further, we manage to avoid the random accesses that incur during the for-loop on Lines 19–20 of Algorithm 2 when the counters for multiple intervals are concurrently updated. Overall, the cost of Smart Counting (excluding sorting) is $O(|R| + |S|)$ due to the constant-time cost of processing at each position of the sweep line.

4 EXPERIMENTAL ANALYSIS

4.1 Setup

For our experiments, we implemented all methods in C++ and compiled them using gcc (v5.2.1). Note that all data (input collections, active sets, interval points list etc.) resided in main memory.

Methods. Besides gapless hash map, the authors in [13] also discussed a lazy optimization for plane-sweep based Algorithm 1, which buffers consecutive start points in list L from the same input (e.g., R). When producing *IJ* results, a single scan over the active set of the other input (i.e., A^S) is performed for the entire buffer. By restricting buffers to fit inside L1 cache or even the cache registers, this technique reduces cache misses. To enhance *ICSJ* computation, we applied this lazy optimization on Naïve and Simple Counting. For the latter, we buffer consecutive start points from S allowing us to increase $C[r]$ for each $r \in A^R$ by the buffer size instead of 1 as in Lines 19–20 of Algorithm 2. On the other hand, lazy optimization has no effect on Smart Counting.

Datasets. Table 1 details our 4 real-world experimental datasets. FLIGHTS records domestic flights in USA during January 2016 (<https://www.bts.gov>); valid times indicate the duration of a flight. BOOKS records the transactions at Aarhus public libraries in 2013 (<https://www.odaa.dk>); valid times indicate the periods when a book is lent out. GREEND [10, 13] records power usage information in households across Austria and Italy from January 2010 to October 2014; valid times indicate the period of a measurement. WEBKIT records the file history in the git repository of the Webkit project from 2001 to 2016 (<https://webkit.org>); valid times indicate the periods when a file did not change.

Tests. We ran interval count semi-joins using a uniformly sampled subset of each dataset as outer input R and the entire dataset as inner S ; for this purpose, we varied ratio $|R|/|S|$ inside $\{0.25, 0.5, 0.75, 1\}$. To assess the performance of the methods, we measured their total execution time which breaks down to the time spent (i) to generate and sort the list of interval points L , denoted by Sorting, and (ii) to compute the *ICSJ* result, denoted by Joining.

4.2 Experiments

Figures 2 and 3 report the results of our experimental analysis. In specific, Figure 2 reports the total execution time of each method

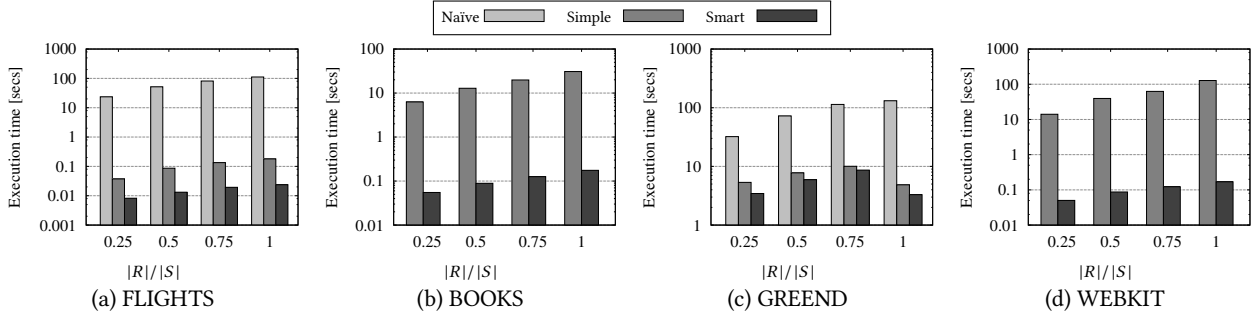


Figure 2: Total execution time while varying the $|R|/|S|$ ratio.

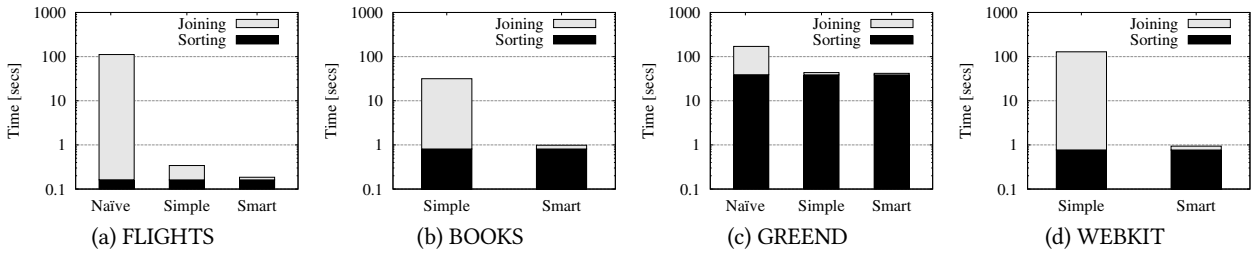


Figure 3: Execution time breakdown for $|R| = |S|$.

while varying the $|R|/|S|$ ratio and Figure 3 reports a breakdown of the execution time for the $|R| = |S|$ case. As expected, we were able to run the Naïve method only when $ICSJ$ was very selective, i.e., for datasets FLIGHTS and GREEND. Recall from Section 3 that Naïve first evaluates the IJ of the input collections; on BOOKS and WEBKIT, it was impossible to accommodate the enormous number of IJ result pairs in main memory.⁴

Figure 2 demonstrates the efficiency of the Smart Counting approach, which outperforms Simple Counting in all cases. In fact, Simple is competitive to Smart only for very selective join setups (see Figure 2(c)) while in all other cases, Smart is at least one order of magnitude faster. To explain the performance cost differences between Smart and Simple, we breakdown their execution times. In Figure 3(c), the execution cost of Simple is dominated by the generation and sorting of the points list L , because the number of overlapping interval pairs is small, rendering the inner loop at lines 19-20 of Algorithm 2 cheap. On other hand, when there is a large number of overlapping intervals, Figure 3 unveils that maintaining active sets and performing random accesses to update C counters severely impacts the joining time of Simple. In contrast, observe that the cost by Smart to compute the $ICSJ$ result is always much lower than that of generating/sorting L . This is expected because Smart is insensitive to the IJ result, as discussed in Section 3.2.

5 CONCLUSIONS

In this paper, we studied the evaluation of the interval count semi-join operation; we presented an efficient algorithm based on plane sweep. Our algorithm has lower complexity compared to state-of-the-art interval join algorithms if the number of join results is large. We experimentally showed that its overhead on top of sorting the data is minimal in all setups, which means that it is especially tailored in cases where the join inputs are already sorted (e.g., in streaming data applications). In the future, we plan to further study the semantics and the evaluation of top- k

⁴We also experimented with a version of Naïve that flushes the IJ result pairs on disk which was even slower.

interval joins. We also intend to investigate the applications of interval joins and other temporal operations in streaming data.

ACKNOWLEDGEMENTS

This work was partially funded by the European Union’s Horizon 2020 research and innovation programme under grant agreement No 657347.

REFERENCES

- [1] Panagiotis Bouros and Nikos Mamoulis. 2017. A Forward Scan based Plane Sweep Algorithm for Parallel Interval Joins. *PVLDB* 10, 11 (2017), 1346–1357.
- [2] Francesco Cafagna and Michael H. Böhlen. 2017. Disjoint interval partitioning. *Vldb J.* 26, 3 (2017), 447–466.
- [3] Bhupesh Chawda, Himanshu Gupta, Sumit Negi, Tanveer A. Faruque, L. Venkata Subramaniam, and Mukesh K. Mohania. 2014. Processing Interval Joins On Map-Reduce. In *EDBT*.
- [4] Reynold Cheng, Sarvjeet Singh, Sunil Prabhakar, Rahul Shah, Jeffrey Scott Vitter, and Yuni Xia. 2006. Efficient join processing over uncertain data. In *CIKM*.
- [5] Anton Dignós, Michael H. Böhlen, and Johann Gamper. 2014. Overlap interval partition join. In *SIGMOD*.
- [6] Jost Enderle, Matthias Hampel, and Thomas Seidl. 2004. Joining Interval Data in Relational Databases. In *SIGMOD*.
- [7] Jaewoo Kang, Jeffrey F. Naughton, and Stratis Viglas. 2003. Evaluating Window Joins over Unbounded Streams. In *ICDE*.
- [8] Martin Kaufmann, Amin Amiri Manjili, Panagiotis Vagenas, Peter M. Fischer, Donald Kossmann, Franz Färber, and Norman May. 2013. Timeline index: a unified data structure for processing queries on temporal data in SAP HANA. In *SIGMOD*.
- [9] T. Y. Cliff Leung and Richard R. Muntz. 1992. Temporal Query Processing and Optimization in Multiprocessor Database Machines. In *VLDB*.
- [10] Andrea Monacchi, Dominik Egarter, Wilfried Elmenreich, Salvatore D’Alessandro, and Andrea M. Tonello. 2014. GREEND: An energy consumption dataset of households in Italy and Austria. In *SmartGridComm*. 511–516.
- [11] Jack A. Orenstein. 1986. Spatial Query Processing in an Object-Oriented Database System. In *SIGMOD*.
- [12] Danila Piatov and Sven Helmer. 2017. Sweeping-Based Temporal Aggregation. In *SSTD*. 125–144.
- [13] Danila Piatov, Sven Helmer, and Anton Dignós. 2016. An interval join optimized for modern hardware. In *ICDE*.
- [14] Arie Segev and Himawan Gunadhi. 1989. Event-Join Optimization in Temporal Relational Databases. In *VLDB*.
- [15] Cheng Sheng, Yufei Tao, and Jianzhong Li. 2012. Exact and approximate algorithms for the most connected vertex problem. *ACM TODS* 37, 2 (2012), 12:1–12:39.
- [16] Manli Zhu, Dimitris Papadias, Jun Zhang, and Dik Lun Lee. 2005. Top-k Spatial Joins. *IEEE TKDE* 17, 4 (2005), 567–579.

Notable Characteristics Search through Knowledge Graphs

Davide Mottin¹, Bastian Grasnack², Axel Kroschek², Patrick Siegler², Emmanuel Müller¹

Hasso Plattner Institute

¹first.last@hpi.de ²first.last@student.hpi.de

ABSTRACT

Query answering routinely employs knowledge graphs to assist the user in the search process. Given a knowledge graph that represents entities and relationships among them, one aims at complementing the search with intuitive but effective mechanisms. In particular, we focus on the comparison of two or more entities and the detection of unexpected, surprising properties, called *notable characteristics*. Such characteristics provide intuitive explanations of the peculiarities of the selected entities with respect to similar entities. We propose a solid probabilistic approach that first retrieves entity nodes similar to the query nodes provided by the user, and then exploits distributional properties to understand whether a certain attribute is interesting or not. Our preliminary experiments demonstrate the solidity of our approach and show that we are able to discover notable characteristics that are indeed interesting and relevant for the user.

1 INTRODUCTION

Search engines have greatly evolved from simple indexes of pages to complex systems that are able to predict user intentions and answer queries on a variety of data sources. One way to improve the search quality is by using a knowledge graph that represents entities (e.g., Angela Merkel, Germany) as nodes and relationships between them (e.g., leaderOf) as edges in a graph. The great expressiveness of knowledge graphs can complement the search with more flexible search paradigms. Assume for instance a scholar who requires to know some non-trivial facts about Angela Merkel and Emmanuel Macron with respect to other country leaders. It would be interesting to discover for instance that Angela Merkel studied Physics as opposed to most of the other leaders, and that she has no children. We call this fact a *notable characteristic*, to remark the unexpected and non-trivial aspect of the discovery. To this end, we propose a novel type of search called *notable characteristics search* that allows the retrieval of such facts from a set of input query entities. Discovering notable characteristics constitutes a ground for targeted analyses of products (e.g., comparing two cameras effectively) in electronic commerce or microorganisms in biological networks (e.g., two influence bacteria) with respect to a set of similars. As a consequence, in all the cases in which a knowledge graph is available, the discovery of notable characteristics becomes an expressive and powerful search type for any user, from experts and practitioners to novice users.

In our setting, we assume the user provides a set of *query* nodes to be compared and the algorithm finds a set of notable characteristics of these nodes. Given a node, a property is a relationship with other nodes (e.g., leaderOf). A characteristic or property is notable, if it deviates from what one would expect for the kind of nodes (e.g., presidents) into consideration. To the best of our knowledge, this is the first study of automatic discovery of notable characteristics (or properties).

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

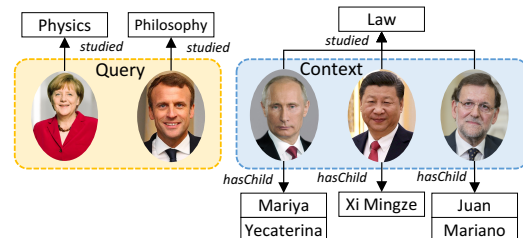


Figure 1: An example knowledge graph, the query (Merkel and Macron), and the discovered context nodes (Putin, Xi Jinping, and Rajoy). The fact that Merkel and Macron do not have children is a notable characteristic.

The discovery of notable characteristics entails two challenges. First, given the set of query nodes we need to compare them to only those nodes that are similar to some extent. Second, we need to select only those properties that are significantly different from the one expressed in the query. Note that tackling the first challenge is very important, as the comparison of the query nodes has to be performed with a set of similar nodes, which we call the context of the query. Consider the naïve approach that returns notable characteristics simply by comparing the query nodes and assume that the user provides “Angela Merkel” and “Theresa May” as query. This is a counter example for the naïve direct comparison, as it will not return the gender as a notable characteristic. Both query nodes are female, however only in comparison with other presidents this becomes an interesting fact. On the other extreme, selecting all the nodes in the graph as context will mislead the analysis towards non-relevant nodes. Take our example of “Angela Merkel” and “Emmanuel Macron”. A naïve selection of all humans will not work as context, since the gender characteristic is not notable among all persons.

It is crucial to provide a thorough context selection to prevent the above cases. Therefore, we introduce the discovery of *context nodes*, i.e., nodes similar to the query nodes. An example of the proposed approach is depicted in Figure 1. To this end, we devise a method that exploits metapaths [12] and random walks for context discovery. We also propose a generic framework that efficiently discover notable characteristics through a novel probabilistic approach based on distribution comparison.

Our contributions are summarized as follows: (1) We formalize the problem of *notable characteristics search* given a set of query nodes as input. (2) We show how to effectively compute metapaths to find the context nodes in knowledge graphs. (3) We introduce a probabilistic approach to discover notable characteristics given a query node set. (4) We experimentally evaluate our context selection approach through a user study, and show evidence of our discovered notable characteristics and the real time performance of the proposed algorithms.

2 NOTABLE CHARACTERISTICS SEARCH

We are given a set \mathcal{A} of node labels and a set \mathcal{L} of edge labels. A *knowledge graph* is a directed graph $G : \langle \mathcal{V}, \mathcal{E}, \phi, \psi \rangle$, where \mathcal{V} is a set of nodes, $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is a set of edges, $\phi : \mathcal{V} \mapsto \mathcal{A}$, $\psi : \mathcal{E} \mapsto \mathcal{L}$ are node and edge labeling functions, respectively.

For simplicity, we assume that everything is modeled as relationships and nodes. This is the case for attributes such as birth date: we assume that the date itself is a node connected with a birthdate relationship. Additionally, we assume that for every edge $e \in \mathcal{E}$ with type $\psi(e) = l$ exists a reverse edge e^{-1} with $\psi(e^{-1}) = l^{-1}$ to model cases such as `presidentOf` and `hasPresident`. The above assumptions do not change the generality of the methods but simplify the notation and the analysis.

We aim at discovering *notable characteristics* expressed as a set of input query nodes (entities) in relation to their similars. This intuitive definition entails two questions: (1) what is the set of similars? (2) what are the notable characteristics?

Given a knowledge graph $G : \langle \mathcal{V}, \mathcal{E}, \phi, \psi \rangle$, the set of input nodes, referred to as *query set* or query in short, is any set $Q \subseteq \mathcal{V}$. The query is manually provided by the user and therefore considered reasonably small (i.e., ≤ 10 elements). The first question concerns the definition of a set of similars referred in this work as *context nodes*. We assume the existence of a *similarity function* $\sigma : \mathcal{V} \times 2^{\mathcal{V}} \mapsto \mathbb{R}$ that assigns a high score to nodes that are similar to those in the query set and low otherwise. Given such similarity, the context are the top- k most similar nodes.

Definition 2.1 (Context set). Given a knowledge graph $G : \langle \mathcal{V}, \mathcal{E}, \phi, \psi \rangle$, a query set $Q \subseteq \mathcal{V}$, a similarity function $\sigma : \mathcal{V} \times 2^{\mathcal{V}} \mapsto \mathbb{R}$, and a parameter k , the *context set* (or simply context) is a set $C \subseteq \mathcal{V}$ such that $Q \cap C = \emptyset$, $|C| = k$, and for each $n_c \in C \wedge n \in \mathcal{V} \setminus (Q \cup C)$, $\sigma(n, Q) \leq \sigma(n_c, Q)$.

The second question concerns the notable characteristics. The characteristics are attributes or relationships of a specific node since they implicitly represent a signature of the node itself. We assume the existence of a generic *discrimination function* $\delta : \mathcal{L} \times 2^{\mathcal{V}} \times 2^{\mathcal{V}} \mapsto \mathbb{R}_0^+$, which represents how a specific characteristic is discriminative or unexpected comparing two set of nodes. The discrimination function returns 0 if the value is not discriminative. We are now ready to define a notable characteristic.

Definition 2.2 (Notable characteristic). Given a knowledge graph $G : \langle \mathcal{V}, \mathcal{E}, \phi, \psi \rangle$, a query $Q \subseteq \mathcal{V}$, a context $C \subseteq \mathcal{V}$, and a discrimination function $\delta : \mathcal{L} \times 2^{\mathcal{V}} \times 2^{\mathcal{V}} \mapsto \mathbb{R}_0^+$ a *notable characteristic* is a relationship $l \in \mathcal{L}|_{Q \cup C}$ such that $\delta(l, Q, C) \neq 0$.

The notation $\mathcal{L}|_{Q \cup C} = \{l \mid \exists x \in Q \cup C, y \in \mathcal{V} \text{ s.t. } (x, y) \in \mathcal{E} \wedge \psi(x, y) = l\}$ denotes the set of edge labels restricted to those found in the edges directly connected to $Q \cup C$.

The general problem we aim to solve is efficiently returning the notable characteristics, given a query, a similarity function and a discrimination function.

PROBLEM 1 (NOTABLE CHARACTERISTICS SEARCH). *Given a knowledge graph $G : \langle \mathcal{V}, \mathcal{E}, \phi, \psi \rangle$, a query $Q \subseteq \mathcal{V}$, a similarity function $\sigma : \mathcal{V} \times 2^{\mathcal{V}} \mapsto \mathbb{R}$ and a discrimination function $\delta : \mathcal{L} \times 2^{\mathcal{V}} \times 2^{\mathcal{V}} \mapsto \mathbb{R}_0^+$, find the set of notable characteristics.*

3 A PROBABILISTIC SOLUTION

The problem entails the definition of appropriate σ (similarity) and δ (discrimination) functions. Section 3.1 introduces a graph-principled solution based on random walks for retrieving context nodes, while Section 3.2 describes a probabilistic approach to effectively discover notable characteristics.

3.1 Finding the context

Given the query Q , we define a similarity function σ to retrieve a set of context nodes. Although many notions of similarity functions have been developed, such as SimRank [4], none seems suitable to our case since they compare two nodes at the time. We devise an algorithm that takes into account edge labels and combines the advantages of random walk and metapath approaches.

In the random walk model, a walker chooses one of the outgoing edges from a node with uniform probability. Motivated by information theoretic notions applied to graphs [10], instead of uniform probability, we favor edge labels with lower frequency. We define $\mathcal{E}_l = \{(i, j) \in \mathcal{E} \mid i, j \in \mathcal{V}, \psi(i, j) = l\}$, the set of edges having label $l \in \mathcal{L}$. The frequency of a label l is the fraction of l -labeled edges with respect to the total number of edges. The weighted adjacency matrix is a $|\mathcal{V}| \times |\mathcal{V}|$ matrix, where for each node i and j , $A_{ij} = 1 - |\mathcal{E}_l|/|\mathcal{E}|$ if $(i, j) \in \mathcal{E}$ and $A_{ij} = 0$ otherwise.

The Personalized PageRank is the vector $\mathbf{p} = c\tilde{A}\mathbf{p} + (1 - c)\mathbf{v}$, where $\tilde{A}_{ij} = A_{ji}/\sum_k A_{jk}$, c is the damping factor, and \mathbf{v} is vector called personalization vector. In our experiments the damping factor is 0.8, in line with previous works. We compute \mathbf{p} starting from each node in the query to retrieve the k nodes with the highest score. This is done by setting $\mathbf{v}_n = 1/|Q|$ for each $n \in Q$. We refer to this baseline as RANDOMWALK.

The PageRank disregards which type of relationships are involved in the random walk, discarding the valuable information encoded in the surrounding of the query nodes. To this end, we adopt the notion of metapath [8, 12] which generalizes the concept of path. A metapath for a path $\langle n_1, \dots, n_t \rangle$, $n_i \in \mathcal{V}$, $1 \leq i \leq t$ is a sequence $\langle \phi(n_1), \psi(n_1, n_2), \dots, \psi(n_{t-1}, n_t), \phi(n_t) \rangle$ that alternates node and edge labels along the path.

We mine metapaths as follows. We sample a node in $\mathcal{V} \setminus Q$ uniformly and run a random walk until a query node is reached. The sequence of edge labels m encountered in the random walk is added to the set of metapaths M along with the number of times $c(m)$ the same metapath has been found so far. It has been proved that random walks are effective in mining metapaths [7].

Once the metapaths are retrieved, we compute a score for each node based on the probability that some metapath starting from a query node ends in such node. Given the set of metapaths M , $\{n \overset{m}{\rightsquigarrow} n'\}$ is the set of paths from node n to n' matching metapath $m \in M$. The score of a node $n' \in \mathcal{V} \setminus Q$ with respect to $n \in Q$ is

$$\sigma(n', Q) = \sum_{m \in M, n \in Q} \frac{|\{n \overset{m}{\rightsquigarrow} n'\}|}{|\{n \overset{m}{\rightsquigarrow} n'' \mid n'' \in \mathcal{V} \setminus Q\}|} \Pr(m) \quad (1)$$

$\Pr(m) = c(m)/\sum_{m \in M} c(m)$ is the probability of choosing metapath m . Intuitively, σ gives a higher score to nodes that are reachable through frequent metapaths connecting the query nodes or connected through many of these metapaths. Hence, nodes that are reached from infrequent metapaths will have a low score. We refer to this method as CONTEXTRW.

3.2 Comparing the distributions

We revise the definition of notable characteristics in probabilistic terms. Assume we have computed the distribution of values for each characteristic (i.e., edge label) for both query and context nodes found with the method in Section 3.1. Such distribution of the context represents the expected, or *normal* behavior, to be evaluated against the *notable* behaviour of the query set.

Formally, for each characteristic $l \in \mathcal{L}$, we consider two vectors in order to evaluate its notability. The first represents the count of the node labels (e.g., France) connected to a specific edge label (e.g., `bornIn`). This expresses information about the values in the nodes and can be used to identify cases where different attribute values are relevant. For instance, in the query in Figure 1 all people are European, while in the context half are Europeans and half Asian. We refer to these vectors as *instance vectors*

$$\mathcal{I}_q(l, C, Q) = (x_1, x_2, \dots, x_t), \mathcal{I}_c(l, C) = (y_1, y_2, \dots, y_t)$$

where x_i and y_i are the number of occurrences of node i at the end of an edge labeled l from a node in Q and C , respectively. In the example in Figure 1, $\mathcal{I}_q(\text{studied}, C, Q) = (1, 1, 0)$,

$\mathcal{I}_c(\textit{studied}, C) = (0, 0, 3)$, where the positions in the vector indicate (Physics, Philosophy, Law). Note that both vectors have the same size, so x_i is zero if i appears only in the context.

Similarly, the second vector represents aggregates over the number of occurrences of a specific edge label in the context, which are useful to represent the characteristic ‘‘Angela Merkel’’ has no child, instead of listing the children names. We refer to these vectors as *cardinality vectors*.

$$\mathcal{K}_q(l, C, Q) = (x_1, x_2, \dots, x_l), \mathcal{K}_c(l, C) = (y_1, y_2, \dots, y_l)$$

where x_i and y_i are the number of times a node in Q and C respectively has i edges labeled l .

Both vectors can be built by iterating through the nodes in each set and counting the respective occurrences. For a given $l \in \mathcal{L}$, this results in two scores $\delta_{\mathcal{I}}$ and $\delta_{\mathcal{K}}$. The final score δ is the maximum score between $\delta_{\mathcal{I}}$ and $\delta_{\mathcal{K}}$.

$$\delta(l, C, Q) = \max(\delta_{\mathcal{I}}(l, C, Q), \delta_{\mathcal{K}}(l, C, Q)) \quad (2)$$

Many measures have been proposed in statistics to compare two vectors in terms of distributions, such as the Kullback-Leibler (KL) divergence, the χ^2 test, and Earth Mover’s Distance (EMD). However, most of them draw specific assumptions, such as non-zero probabilities, or normality, that are not fulfilled in our case since \mathcal{I} and \mathcal{K} have no natural ordering and no distance-function between the values. Therefore, we resort to a more natural multinomial test that better expresses the relationship between our distributions. The multinomial test assumes that a set of observations (the query) is drawn from a multinomial distribution (the context). If the values observed in the query are drawn from the multinomial, than the hypothesis cannot be rejected and the characteristic is marked as non-notable; otherwise, the success of the test denotes that the characteristic is notable.

Assume we have a random variable $X_{N, \pi} \sim \text{Mult}(N, \pi)$, with parameters N and distribution π . We normalize \mathcal{I}_c and \mathcal{K}_c to express the probability distributions $\widehat{\mathcal{I}}_c = \mathcal{I}_c / \|\mathcal{I}_c\|_1$ and $\widehat{\mathcal{K}}_c = \mathcal{K}_c / \|\mathcal{K}_c\|_1$. The significance probability is

$$\Pr_s(X_{N, \pi} = x) = \sum_{y: \Pr(X_{N, \pi} = y) \leq \Pr(X_{N, \pi} = x)} \Pr(X_{N, \pi} = y)$$

where $\Pr_s(\pi, x)$ is the probability of x or any equally or less likely outcome being drawn from the probability distribution¹. A difference in distributions is considered significant if the hypothesis is rejected with probability $p > 0.95$.

$$\text{MT}(\pi, x) = \begin{cases} 1 - \Pr_s(X_{N, \pi} = x) & \text{if } \Pr_s(\dots) \leq 0.05 \\ 0 & \text{otherwise} \end{cases}$$

Finally, δ id defined as $\delta_{\mathcal{I}}(l, C, Q) = \text{MT}(\widehat{\mathcal{I}}_c(l, C), \mathcal{I}_q(l, C, Q))$ and $\delta_{\mathcal{K}}(l, C, Q) = \text{MT}(\widehat{\mathcal{K}}_c(l, C), \mathcal{K}_q(l, C, Q))$.

4 EXPERIMENTAL EVALUATION

We experimentally evaluate our approach on different datasets and show the impact of the parameters on the final results.

Datasets: We perform experiments on two real datasets.

- YAGO is a large knowledge graph based on Wikipedia, Wordnet and Geonames, with 3.3M nodes, 27M edges, 366K node types and 38 edge labels. We downloaded YAGO 2.5² and converted node attributes to edges and attribute values to node labels.
- Lmdb³: LinkedMDB is a knowledge graph for the movie domain, extracted from the Internet Movie Database (IMDB), with 739K nodes, 1.6M edges, and 18 edge types.

Experimental Setup: We implemented our solution in Java 1.8, and ran the experiments on a Intel i5-4210U 1.7 GHz machine

¹In case of large N , an approxiamte Montecarlo sampling is performed.

²<http://resources.mpi-inf.mpg.de/yago-naga/yago2.5>

³<https://datahub.io/dataset/linkedmdb>

with 12GB RAM. All datasets are loaded into Apache Jena triple store. Along our CONTEXTRW described in Section 3.1 for context selection and FINDNC for notable characteristics identification on top of CONTEXTRW, we implement RANDOMWALK, a baseline for context selection based on Personalized PageRank (see Section 3.1) computed through the power iteration method with 10 iterations and $c = 0.8$.

4.1 Evaluating Context selection

We compare the effectiveness of CONTEXTRW with the baseline RANDOMWALK within different topics. Since no ground truth for finding context nodes given a set of query nodes were available, we generated context nodes via CrowdFlower (<https://www.crowdfunder.com>) for 15 query sets in three domains, namely *politicians*, *actors*, and *movie contributors*. For each domain we manually determined 6 entities belonging to the domain and generated queries of increasing size (up to 6 entities). We asked 34 workers to provide a ranked list of related entities given the query, resulting in 7’650 entities. From such entities we removed those mentioned only once and obtained 36 to 76 entities per query that are mapped into YAGO.

Context size $|C|$. Context size affects the quality of the results, since more context nodes potentially lead to better recall but worse precision. Figure 2a compares RANDOMWALK and our CONTEXTRW in terms of F_1 score at different $|C|$. In all cases, CONTEXTRW performs up to four times better than the RANDOMWALK, vindicating the effectiveness of our metapath constrained random walk in finding context nodes, while RANDOMWALK mostly returns nodes that are close to the query nodes, but semantically irrelevant. Quality does not improve for $|C| > 100$ due to a loss in precision. We also experience a lower variance for CONTEXTRW that exploits metapaths for guiding the search.

Query size $|Q|$. We analyze the performance of the algorithms varying the query size $|Q|$. Figure 2 shows that CONTEXTRW improves in result quality when more query nodes are considered, which means that our method capture semantic relationships between the nodes. On the contrary, RANDOMWALK is not affected by the size of the query disregarding metapaths.

Figure 3a reports the time to compute the context, showing that RANDOMWALK is on average up to two orders of magnitude slower than CONTEXTRW, for $|Q| = 5$. Expectedly, CONTEXTRW is faster with larger queries (<20s comprising the database time), since the chance to end up in a query node is larger.

Figure 2c reports the maximum F_1 of CONTEXTRW at increasing $|Q|$, comparing YAGO and Lmdb datasets within the *actors* domain. Unsurprisingly, CONTEXTRW performs moderately better than YAGO in Lmdb due to the specificity of the dataset; however, YAGO result testifies the generalization ability of CONTEXTRW in larger, more complex datasets.

Number of paths $|M|$. The CONTEXTRW algorithm depends on the number of paths. Figure 2d shows the F_1 score in relation to the context size and the number of paths. The number of paths does not affect the score; however, as shown in Figure 3b the time increases as the length of the metapaths (and also the number, not reported) increases. Therefore, a reasonable choice for the number of metapaths $|M|$ and maximum length is 5.

4.2 Distribution Comparison

Test cases. We show preliminary evidence of the effectiveness of FINDNC with respect to the RANDOMWALK baseline for context retrieval with the multinomial test. The test case in Figure 4a shows the instance distribution of the query $Q = \{\textit{George Clooney},$

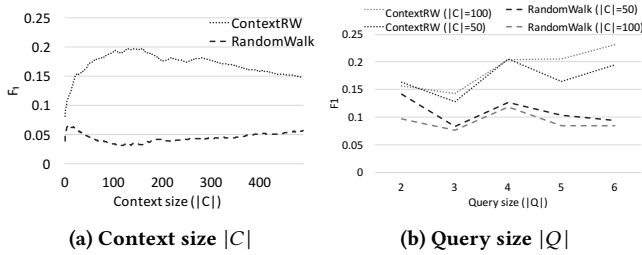


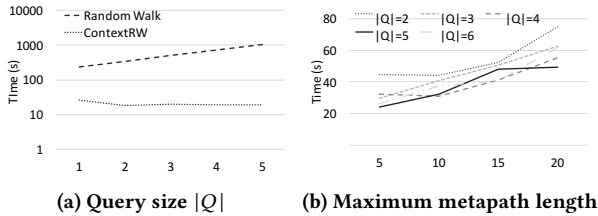
Figure 2: Average quality in terms of F_1 varying parameters $|C|$, $|q|$, and $|M|$ in YAGO dataset.

$ Q $		max F_1	$ C $
2	YAGO	0.23	23
	LMDB	0.30	101
3	YAGO	0.2	107
	LMDB	0.25	122
4	YAGO	0.19	130
	LMDB	0.24	124
5	YAGO	0.25	162
	LMDB	0.26	198
6	YAGO	0.22	285
	LMDB	0.25	139

$ C $	Number of paths ($ M $)			
	5	10	15	20
50	0.15	0.16	0.13	0.15
100	0.22	0.21	0.21	0.21
150	0.22	0.23	0.23	0.23
200	0.22	0.22	0.22	0.22

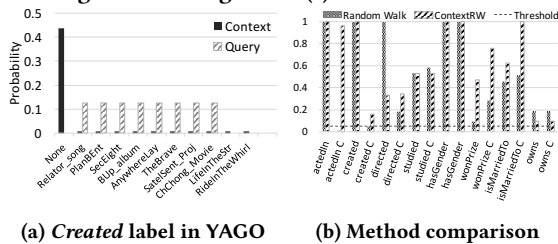
(c) Dataset comparison

(d) Number of paths $|M|$



(a) Query size $|Q|$ (b) Maximum metapath length

Figure 3: Average time (s) in YAGO dataset.



(a) Created label in YAGO

(b) Method comparison

Figure 4: Test cases for the actors domain and 5 query nodes. A “C” in the labels denotes cardinality distributions

Brad Pitt, Leonardo DiCaprio, Scarlett Johansson, Johnny Depp over the top-100 context nodes for the *created* edge label. The *created* edge label is absent in 43% cases (represented as *None* instance), whereas all the other values are equally likely with 0.66% chances. The query presents a different distribution, with one actor without *created* labels and all the others with a different value. This clear deviation from the context is a notable characteristic by the multinomial test.

In the second test case, not reported due to space limits, we test the query $\{Douglas Adams, Terry Pratchett\}$ against the top-30 nodes as context. Our solution identified the edge *influences* as a notable characteristic. This is because the two authors in the query influenced an actor that was influenced by only 3 in total.

Algorithm comparison. Figure 4b compares FINDNC with RW-MULT, with query $\{George Clooney, Brad Pitt, Leonardo DiCaprio, Scarlett Johansson and Johnny Depp\}$. All items above the threshold, depicted as a dashed line, are considered not interesting ($\delta = 0$). The random walk selects mostly famous people in the movie business; hence, *actedIn* that connects actors with movies, is rare in the context but common in the query. However, this is clearly not correct and our FINDNC algorithm marks *actedIn* as uninteresting. Similarly, *hasWonPrize* shows a significant difference between the two algorithms, as winning a prize is common for actors (75%), but not so in the rather mixed random walk context (only 25%). The chart also shows that the significance level of the multinomial test can be used as a parameter to obtain the desired “interestingness” level. Choosing 0.1 would include the *owns* relationship as a notable characteristic, revealing that Brad Pitt is (according to the dataset) the only relevant actor to own a company (Plan B Entertainment).

5 RELATED WORK

Finding notable characteristics reminisces the problem of anomaly detection in attributed graphs [1]; yet, it is fundamentally

different, for it does not provide explanations on the nodes returned as anomalies, nor such approaches are query-driven.

Node comparison measures. Node similarities, typically defined in terms of neighbors, is a centerpiece for community detection, classification, and link prediction. Structural equivalence, such as SimRank [4] defines two nodes similar if the neighbors are similar. Random walk approaches, such as Personalized PageRank [2] and HITS [5] can also be used to find structurally similar nodes. However, node similarities cannot readily explain differences among query nodes and other similar nodes.

Seed set expansion. Seed set expansion, or example-based methods, refers to methods that ask the user to provide an initial set of entities or structures and retrieve similar nodes. Seed nodes are used to discover groups of nodes with similar characteristics [6] exploiting the specificity of each node in the seed set. Likewise, seed-based approaches are used to discover dense graph regions [3, 11]. Although these methods provide multiple groups of nodes they cannot properly explain the characteristics and the differences among them; in general, they do not directly compare the query nodes with the others.

Relevant path summarization. Our problem is connected to the discovery of metapaths between nodes [8, 12]. Methods have been proposed to automatically learn metapaths from a given seed set [9]. However, metapaths cannot express the lack of an edge (e.g., Angela Merkel has no children), nor they cannot detect notable characteristics: Being born in the same place is notable, only if similar people are born in different places.

REFERENCES

- [1] Leman Akoglu, Hanghang Tong, and Danai Koutra. 2015. Graph based anomaly detection and description: a survey. *DAMI* 29, 3 (2015), 626–688.
- [2] Soumen Chakrabarti. 2007. Dynamic personalized pagerank in entity-relation graphs. In *WWW*. 571–580.
- [3] Aristides Gionis, Michael Mathioudakis, and Antti Ukkonen. 2015. Bump hunting in the dark: Local discrepancy maximization on graphs. In *ICDE*. 1155–1166.
- [4] Glen Jeh and Jennifer Widom. 2002. SimRank: a measure of structural-context similarity. In *KDD*. 538–543.
- [5] Jon M Kleinberg. 1999. Authoritative sources in a hyperlinked environment. *JACM* 46, 5 (1999), 604–632.
- [6] Isabel M Kloumann and Jon M Kleinberg. 2014. Community membership identification from small seed sets. In *KDD*. 1366–1375.
- [7] Sangkeun Lee, Sanghyeb Lee, and Byoung-Hoon Park. 2015. PathMining: A Path-Based User Profiling Algorithm for Heterogeneous Graph-Based Recommender Systems.. In *FLAIRS Conference*. 519–523.
- [8] Sangkeun Lee, Sungchan Park, Minsuk Kahng, and Sang-goo Lee. 2012. Pathrank: a novel node ranking measure on a heterogeneous graph for recommender systems. In *CIKM*. 1637–1641.
- [9] Changping Meng, Reynold Cheng, Silviu Maniu, Pierre Senellart, and Wangda Zhang. 2015. Discovering meta-paths in large heterogeneous information networks. In *WWW*. 754–764.
- [10] Davide Mottin, Matteo Lissandrini, Yannis Velegrakis, and Themis Palpanas. 2016. Exemplar queries: a new way of searching. *Vldb J.* 25, 6 (2016), 741–765.
- [11] Natali Ruchansky, Francesco Bonchi, David Garcia-Soriano, Francesco Gullo, and Nicolas Kourtellis. 2015. The Minimum Wiener Connector Problem. In *SIGMOD*. 1587–1602.
- [12] Yizhou Sun, Jiawei Han, Xifeng Yan, Philip S Yu, and Tianyi Wu. 2011. Pathsim: Meta path-based top-k similarity search in heterogeneous information networks. *PVLDB* 4, 11 (2011), 992–1003.

EmbedS: Scalable, Ontology-aware Graph Embeddings

Gonzalo I. Diaz
 Department of Computer Science
 University of Oxford
 gonzalo.diaz@cs.ox.ac.uk

Achille Fokoue
 IBM T. J. Watson Research Center
 achille@us.ibm.com

Mohammad Sadoghi
 Exploratory Systems Lab
 University of California, Davis
 msadoghi@ucdavis.edu

ABSTRACT

While the growing corpus of knowledge is now being encoded in the form of *knowledge graphs* with rich semantics, the current graph embedding models do not incorporate ontology information into the modeling. We propose a scalable and ontology-aware graph embedding model, EmbedS, which is able to capture RDFS ontological assertions. EmbedS models entities, classes, and properties differently in an RDF graph, allowing for a geometrical interpretation of ontology assertions such as type inclusion, subclassing, and alike.

1 INTRODUCTION

A growing corpus of knowledge is being encoded in the form of *knowledge graphs*, i.e. in the form of (s, p, o) triples which represent simple subject-predicate-object sentences. These triple-based formats consist of binary relational data, where an (s, o) pair is effectively declared to be related by the p binary relation. Although more complex—or higher arity—information is difficult to express in this way, the simple syntax enjoyed by these triple-based data models has proved highly useful for a wide range of applications and has thus been widely adopted. Today, the linked open data cloud consists of hundreds of interlinked datasets, including a few very large knowledge graphs such as Freebase and DBpedia, which contain billions of triples and millions of entities.

Modern knowledge graphs are used in applications such as web search, where graph data provides structured data that complements hyperlink answers, artificial intelligence question answering systems such as Watson and voice-based assistants, and semantic web query engines that run powerful declarative query languages such as SPARQL. However, there are still important issues to be resolved. Even the largest knowledge graphs are extremely incomplete (i.e. many true facts have yet to be encoded as triples) and prone to errors (often triples encoding incorrect facts are included) [12]. The main tasks of link prediction (which consists of predicting new triples) and triplet classification (which seeks to assign a probability that a certain triple—be it new or existing—is true or not) look to address these shortcomings. In this context, there is renewed interest in machine learning over (binary) relational data. The techniques used vary [9] from rule-based learning [5] to tensor factorisation, neural network-based approaches, etc. In this work we focus on latent feature-based techniques, which are also known as graph embeddings.

Graph embeddings correspond to latent feature statistical relational learning models in that they assume the existence of a set of n latent features—or random variables—that account for the predictions we desire (triplet classification or otherwise). As such, these latent features provide machine learning-friendly representations of graph data, which can then be used as input to further

machine learning tools such as neural networks or logistic regression for classification. The input to a graph embedding model, as with other statistical relational learning techniques, is a knowledge graph, and the output is a mapping which associates to each entity in the graph the set of n real-valued latent features. The name *graph embedding* refers to the geometrical interpretation given to the obtained latent features: the entities are interpreted as points in an n -dimensional real coordinate space, and thus are said to have been embedded into said space. On the other hand, the relations of the knowledge graph are varyingly represented as translational vectors [3], matrix transformations [11], etc.

Graph embedding models are usually trained via the minimisation of a global cost function, which can be expressed as the sum of a cost assigned to each triple in the knowledge graph. The minimisation itself is achieved via stochastic gradient descent or similar methods. One of the greatest advantages of graph embedding models, and especially translational models (loosely defined as those models which represent relations as one or more translational vectors, as opposed to transformation matrices), is that they are able to perform efficiently for very large graphs and with high accuracy. Current graph embedding models achieve high performance on basic learning tasks such as link prediction—ranking the correct entity among the top ten candidates 95% of the time [10].

Despite the positive scenario described above, one of the greatest limitations of current graph embedding models is that they consider a very simple model of the data: they consider a knowledge graph to be a set of triples, where each triple mentions two entities and one relation, with no special semantics for any particular triple. In contrast, standard knowledge graphs are often accompanied by rich ontological information which encodes a wealth of metadata, including type hierarchies and other constraints. Current graph models are entirely agnostic to such metadata, and thus ontological triples are usually manually removed before training (or, if they are included, simply interpreted as plain data triples). One of the consequences of this is that current graph embedding models do not directly incorporate constraints that humans would consider obvious (e.g. knowing that a human cannot be friends with a building). A huge potential exists in the rich ontologies that inform modern knowledge graphs, and the objective of this work is to explore ways in which such ontologies can become first class citizens of graph embedding models.

Specifically, we explore the problem of graph embedding on ontology-rich knowledge graphs, where the ontology is specified in a standard language such as RDFS (more expressive languages such as OWL2 have been developed, but will not be considered in this study). Drawing on the geometrical interpretation that graph embeddings give to their latent features (namely, that entities are embedded as points in a real coordinate space), we explore the case where RDFS classes are correspondingly modelled as sets of points in the same coordinate space, and relations are embedded as sets of pairs of points. This generalisation of the basic geometrical interpretation allows for a natural expression of ontological constraints in the global cost function. The result

is a model we name EmbedS, which is able to model RDFS ontological constraints as first-class citizens. Along with providing the precise definitions for the EmbedS cost function, we provide experimental results that show EmbedS to be comparable to state-of-the-art graph embedding techniques when measured on traditional benchmark knowledge graphs, while performing well on a new ontology-rich dataset we have prepared for the purposes of studying the new enriched geometrical interpretation that EmbedS provides.

The preliminary results presented in this paper showcase the potential of extending current graph embedding research to include ontological information, and will hopefully encourage further development of the area. This paper is organised as follows. In Section 2 we introduce necessary mathematical notation and preliminary definitions. In Section 3 we introduce the EmbedS model, its global cost function, and the geometrical interpretation induced on embedded entities and relations. In Section 4 we explain the experimental setting and evaluation metrics, including a discussion of an ontology-rich benchmark dataset we have prepared for testing the EmbedS model. Finally, in Section 5 we present our conclusions and suggest future avenues of work.

2 PRELIMINARIES AND RELATED WORK

Graph embedding models are usually defined on a *knowledge graph*, which is essentially a set of triples, similar to the RDF data model, but lacking specialised features of the latter, such as the precise definition of IRIs, literals, and the semantics associated with certain keyword IRIs. In this section we will introduce necessary definitions, noting the similarities and discrepancies with the RDF data model. We will then introduce needed concepts such as graph embeddings, and finally we will discuss related work.

Let E and P be two mutually disjoint and countably infinite sets of *entities* and *relations*, respectively. A knowledge graph is then defined as a finite set of triples of the form $(s, p, o) \in E \times P \times E$. As such, a knowledge graph can be interpreted as a directed graph with labelled edges, or as a set of binary relations. As this work will seek to allow graph embeddings to operate on more expressive graph data, it is important to note the differences with the full RDF data model. RDF also considers an infinitely countable set of *literals* L , disjoint from E and P , and note that RDF triples are drawn from the more general set $(E \cup P) \times (E \cup P) \cup (E \cup P \cup L)$. For example, note that RDF allows relations to be mentioned in the subject position of a triple, thus blurring the distinction that exists between relations as edge labels and nodes of the graph. This is generally disallowed in the traditional definition of knowledge graph.

The base RDF data model described provides very little semantics other than the basic interpretation of the triple as a fact. To enrich an RDF dataset, several ontology languages have been developed, of which RDFS is one of the simplest. RDFS defines the following core keyword IRIs: `rdf:type`, `rdfs:subClassOf`, `rdfs:subPropertyOf`, `rdfs:domain`, and `rdfs:range` (abbreviated `type`, `sc`, `sp`, `dom`, `range`, `resp.`), which are assigned special semantics in order for richer knowledge—including basic type systems—to be encoded into RDF format¹.

In what follows, we will consider, in addition to the sets E and P , a set C of *classes*, also infinitely countable and disjoint from

the previous two. Furthermore, we define five special relations `type`, `sc`, `sp`, `dom`, `range` $\in P$. We will only consider triples $t = (s, p, o)$ for which one of the following hold:

- $(s, p, o) \in E \times P \times E$ (called data triples),
- $(s, p, o) \in E \times \{\text{type}\} \times C$ (called type triples),
- $(s, p, o) \in C \times \{\text{sc}\} \times C$ (called subclass triples),
- $(s, p, o) \in P \times \{\text{sp}\} \times P$ (called subproperty triples),
- $(s, p, o) \in P \times \{\text{dom}\} \times C$ (called domain triples),
- $(s, p, o) \in P \times \{\text{range}\} \times C$ (called range triples).

An RDF data graph (or, simply, a graph) D is a finite set of triples such that for every triple $t \in D$, t is a data triple or a type triple. An RDFS ontology (or, simply, an ontology) S is a finite set of triples such that for every triple $t \in S$, t is not a data triple or a type triple.

Example 2.1. Consider the graph $D = \{(anne, \text{type}, \text{Woman}), (john, \text{type}, \text{Man}), (john, \text{knows}, anne)\}$. This data about people and their relationships can be enriched with the ontology $S = \{(\text{Woman}, \text{sc}, \text{Person}), (\text{Man}, \text{sc}, \text{Person})\}$. The full dataset is then $I = D \cup S$. Notice that the semantics of RDFS allow us to conclude facts which are not explicitly included in the dataset, such as $(anne, \text{type}, \text{Person})$ and $(john, \text{type}, \text{Person})$. \square

Crucially, the semantics of RDFS allow for inferencing new triples using a series of inference rules. For example, the following is an inference rule for RDFS: $(x, \text{type}, c), (c, \text{sc}, b) \rightarrow (x, \text{type}, b)$, which is read as follows: given a dataset $I = D \cup S$ which contains two triples of the form (a, type, C) and (C, sc, B) , for any entity $a \in E$ and classes B, C , the triple (a, type, B) may be *inferred* to hold [1]. In this context, inferring a triple to hold would cause a query engine to return the inferred triple as an answer to a query.

Graph embedding—and statistical relational learning—models use varying techniques in order to obtain machine-usable representations of knowledge graphs. In general, however, the main problem—that of knowledge base completion—gives a common direction to these techniques: constructing statistical models of the data that allow for link prediction (e.g. given an incomplete fact $(\text{T arantino}, \text{inspiredBy}, ?)$ return the entity that would complete the triple) and triple classification (assign a probability that a triple is true).

The most expressive models involve tensor or matrix factorisation techniques, although these are also the models with the highest complexity, measured in the number of parameters that must be trained. A well-known example of this is RESCAL [11], which explains triples using pairwise interactions of the latent features of entities. Thus, the *cost* associated to a triple x_{ijk} has the form:

$$\text{cost}(x_{ijk}) = \mathbf{e}_i^T \mathbf{W}_k \mathbf{e}_j.$$

Other highly expressive models use techniques such as matrix factorisation, neural tensor networks, and multilayer perceptrons [6, 8]. The latter, also known as word2vec, was strictly a word embedding model, but had as an interesting and unintended consequence a translational property among the latent representation of words: simple binary relations between words could be captured when interpreting the latent representation—*embedding*—of the relation as a *translational vector*. For example, after training on a textual corpus, researchers found that incomplete sentences such as $(\text{Madrid}, \text{capitalof}, \text{Spain})$ had the property that the vector $\mathbf{e}_{\text{Madrid}} + \mathbf{e}_{\text{capitalof}}$ was nearest to $\mathbf{e}_{\text{Spain}}$.

¹The prefixes used are themselves abbreviations. Actually, `rdf:` expands to <http://www.w3.org/1999/02/22-rdf-syntax-ns#> and `rdfs:` expands to <http://www.w3.org/2000/01/rdf-schema#>.

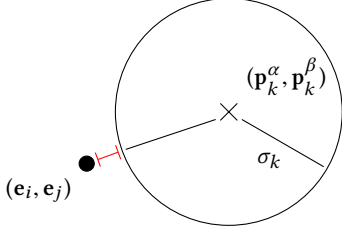


Figure 1: Cost of a triple $(e_i, p_k, e_j) \in I$. The circle represents a $2n$ -sphere of radius σ_k centered at $(\mathbf{p}_k^\alpha, \mathbf{p}_k^\beta)$. The pair (e_i, e_j) is embedded as the $2n$ -dimensional point (e_i, e_j) . The (red) error line shows the cost.

The previous translational property spawned renewed interest in distance-based models that incorporated a *geometrical interpretation* for latent representations. The first of these was TransE [3], and was quickly followed by refinements such as TransH [13], and TransR [7]. While less expressive, translation-based models prove more scalable, as their model complexity is lower and a simpler cost structure allows for more efficient training.

The model proposed in this work draws from research in translation-based models. Our problem scenario focused on ontology-rich knowledge graphs, which contain specific semantics for keyword triples. This metadata has not been considered in previous work, to the best of our knowledge. The idea of considering ontological information in training statistical models, however, has been considered. In [4], type constraints are considered by removing type-constraint-violating triples, improving training speed and link prediction performance. They do not incorporate ontological information into the model as a first class citizen, however, and neither does the geometrical interpretation of the model adapt to reflect the presence of this metadata.

3 MODEL

Consider an RDF dataset $I = D \cup S$, and let $E_I \subseteq E$, $C_I \subseteq C$, and $P_I \subseteq P$ be the sets of entities, classes, and properties that appear in I , respectively. Define for each entity $e_i \in E_I$ an n -dimensional vector of parameters (i.e. variables) $\mathbf{e}_i = (e_{i1}, \dots, e_{in})$; for each class $c_i \in C_I$ define an n -dimensional vector of parameters $\mathbf{c}_i = (c_{i1}, \dots, c_{in})$ and a parameter ρ_i ; and for each property $p_i \in P_I$ define two n -dimensional vectors of parameters $\mathbf{p}_i^\alpha = (p_{i1}^\alpha, \dots, p_{in}^\alpha)$ and $\mathbf{p}_i^\beta = (p_{i1}^\beta, \dots, p_{in}^\beta)$ and a parameter σ_i . We have thus defined a total of $|E_I| \cdot n + |C_I| \cdot (n + 1) + |P_I| \cdot (2n + 1)$ parameters for our model.

For what follows, we first define a distance function **dist** which assigns to every pair of n -dimensional vectors $\mathbf{x} = (x_1, \dots, x_n)$, and $\mathbf{y} = (y_1, \dots, y_n)$ a non-negative real value **dist**(\mathbf{x}, \mathbf{y}), with the standard distance function properties². In this paper, we choose to set **dist**(\mathbf{x}, \mathbf{y}) = $\|\mathbf{x} - \mathbf{y}\|_2 = \sum_{i=1}^n (x_i - y_i)^2$, that is, the L_2 norm of $\mathbf{x} - \mathbf{y}$. We also define an activation function **act**, for which we choose the rectifier function, **act**(x) = $\max(0, x)$.

Assuming that $|E_I| = N_E$, $|C_I| = N_C$, and $|P_I| = N_P$, the cost \mathcal{L} will be a function of all the variables previously defined:

$$\mathcal{L} = \mathcal{L}(\mathbf{e}_1, \dots, \mathbf{e}_{N_E}; \mathbf{c}_1, \dots, \mathbf{c}_{N_C}, \rho_1, \dots, \rho_{N_C}; \mathbf{p}_1^\alpha, \dots, \mathbf{p}_{N_P}^\alpha, \mathbf{p}_1^\beta, \dots, \mathbf{p}_{N_P}^\beta, \sigma_1, \dots, \sigma_{N_P}).$$

We now define the precise form of the cost function \mathcal{L} . For the entire dataset, define $\mathcal{L}^I = \sum_{t \in I} \mathcal{L}_t$, where the cost of each

²(a) **dist**(\mathbf{x}, \mathbf{y}) ≥ 0 , (b) **dist**(\mathbf{x}, \mathbf{y}) = 0 $\Leftrightarrow \mathbf{x} = \mathbf{y}$, (c) **dist**(\mathbf{x}, \mathbf{y}) = **dist**(\mathbf{y}, \mathbf{x}), and (d) **dist**(\mathbf{x}, \mathbf{z}) \leq **dist**(\mathbf{x}, \mathbf{y}) + **dist**(\mathbf{y}, \mathbf{z}).

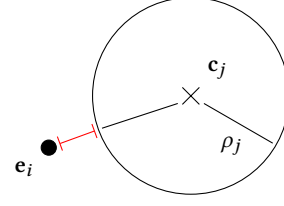


Figure 2: Cost (red error line) of $t = (e_i, \text{type}, c_j)$, where class c_j is embedded as an n -sphere at \mathbf{c}_j with radius ρ_j .

triple $t = (e_i, p_k, e_j) \in I$ (note that $e_i, e_j \in E_I \cup C_I \cup P_I$ and $p_k \in P_I$), is defined as follows:

$$\mathcal{L}_t = \mathbf{act}\left(\mathbf{dist}(\mathbf{e}_i, \mathbf{p}_k^\alpha) + \mathbf{dist}(\mathbf{e}_j, \mathbf{p}_k^\beta) - \sigma_k\right).$$

The geometrical interpretation of this cost is provided in Section 3.1, and is visualized in Figure 1. Next, we define a cost term for each possible RDFS assertion. For each $t \in S$:

- (1) If $t = (e_i, \text{type}, c_j) \in S$, where $e_i \in E_I$ and $c_j \in C_I$, define:

$$\mathcal{L}_t^S = \mathbf{act}\left(\mathbf{dist}(\mathbf{e}_i, \mathbf{c}_j) - \rho_j\right),$$

- (2) If $t = (c_i, \text{sc}, c_j) \in S$, where $c_i, c_j \in C_I$, define:

$$\mathcal{L}_t^S = \mathbf{act}\left(\mathbf{dist}(\mathbf{c}_i, \mathbf{c}_j) - (\rho_j - \rho_i)\right),$$

- (3) If $t = (p_i, \text{sp}, p_j) \in S$, where $p_i, p_j \in P_I$, define:

$$\mathcal{L}_t^S = \mathbf{act}\left(\mathbf{dist}(\mathbf{p}_i^\alpha, \mathbf{p}_j^\alpha) + \mathbf{dist}(\mathbf{p}_i^\beta, \mathbf{p}_j^\beta) - (\sigma_j - \sigma_i)\right),$$

- (4) If $t = (p_i, \text{dom}, c_j) \in S$, where $p_i \in P_I$ and $c_j \in C_I$, define:

$$\mathcal{L}_t^S = \mathbf{act}\left(\mathbf{dist}(\mathbf{p}_i^\alpha, \mathbf{c}_j) - \sigma_i\right),$$

- (5) If $t = (p_i, \text{range}, c_j) \in S$, where $p_i \in P_I$ and $c_j \in C_I$, define:

$$\mathcal{L}_t^S = \mathbf{act}\left(\mathbf{dist}(\mathbf{p}_i^\beta, \mathbf{c}_j) - \sigma_i\right).$$

Finally, we sum, for every triple $t \in S$, the cost of t depending on the RDFS relation it mentions. In that way, we define the cost term $\mathcal{L}^S = \sum_{t \in S} \mathcal{L}_t^S$, and thus define the final cost to be $\mathcal{L} = \mathcal{L}^I + \mathcal{L}^S$.

3.1 Geometrical interpretation

We now give a geometrical interpretation to the model definition. By embedding entities as single n -dimensional vectors we are modelling them as *points* in the n -dimensional euclidean space. Classes, on the other hand, are modelled as *regions* of the euclidean space, being embedded as a vector and a radius, representing n -spheres. This allows for the following geometrical interpretation: if an entity is embedded within the region defined by the embedding of a class, then it is interpreted to be of that type, and vice-versa (see Figure 2). Finally, properties, insofar as they represent binary relations, are modelled as *pairs of points*. Thus, each relation $p_k \in P_I$ has an embedding which consists of two n -dimensional vectors \mathbf{p}_k^α and \mathbf{p}_k^β and a radius ρ_k . The corresponding geometrical interpretation is analogous to the previous case: in $2n$ -space, a pair (e_i, e_j) is interpreted to be related by a relation p_k if the $2n$ -point $(\mathbf{e}_i, \mathbf{e}_j)$ is in the region defined by the $2n$ -sphere centered at $(\mathbf{p}_k^\alpha, \mathbf{p}_k^\beta)$ with radius σ_k (see Figure 1).

The main advantage conferred by this ontology-aware geometrical interpretation is that RDFS classes and ontological assertions are now first-class citizens of the model. By modelling classes as

regions of the euclidean space, for example, certain properties are obtained for free, such as type containment transitivity: if after training the entity `aturing` is (correctly) embedded within the class `Researcher`, and said class is (correctly) embedded fully contained within the region corresponding to class `Person`, then the transitive fact that `aturing` is a `Person` will be provided for free. In this way, the embedding space will presumably encode ontological information geometrically.

4 EXPERIMENTAL EVALUATION

In this section we provide preliminary experimental results showing that EmbedS can perform at state-of-the-art levels on a standard benchmark dataset, while providing a new complementary triple classification method based on the geometrical interpretation which shows encouraging results. An exhaustive experimental evaluation will be left for future work. We use the following datasets for experimental evaluations:

wn18 Dataset extracted from WordNet. This dataset consists of 151,442 triples, 40,943 entities, and 18 relations.

dbpedia32k RDF dataset extracted from DBpedia, consisting of 340,827 triples, 32,657 entities, and 296 relations.

The first dataset has become a standard benchmark in the graph embedding field [3], and allows for an apples to apples comparison between our model and existing work. However, EmbedS has been designed for a fundamentally different problem setting: that of embedding in *ontology-rich* RDF data. In order to test the performance of our model, we have prepared a dataset, named ‘dbpedia32k’ which includes an RDFS ontology.

We now describe the construction of the dbpedia32k dataset. It initially draws from three distinct downloads, which are freely available: the ‘DBpedia Ontology’ file, which contains ontology triples, the ‘Instance Types’ file, which contains data triples of the form (a, type, C) for some entity a and some class C , and the ‘Mappingbased Objects’ file, which contains general data triples, representing facts on entities present in Wikipedia articles. From the Mappingbased Objects file first define a relation to be useful if it appears in at least 1000 triples. We define an entity to be useful if it is mentioned at least 10 times in the file. A uniform sample of useful entities is built, keeping only triples which mention useful relations and these sampled entities only. Finally, we recalculate useful entities (in the filtered dataset) and remove triples which mention non-useful entities. We thus obtain the final set of Mappingbased Objects triples to be used. We complete the dataset by extracting, from Instance Types, triples (a, type, C) where a is mentioned in the previous dataset, and similarly we extract relevant ontology triples from DBpedia Ontology. The resulting complete dataset is split uniformly into three subsets for training, validation, and testing, with proportions 0.8, 0.1, and 0.1, respectively.

Selection of hyperparameters of the model is achieved via random search. Although more sophisticated methods have been proposed, random search is competitive with these systems [2] and thus serves our purposes. For each dataset-model choice (e.g. dbpedia32k with EmbedS), a suitable bounding box for the hyperparameters of the model is chosen, and training is performed over 500 epochs for 1,000 different random hyperparameter values.

To measure the performance of the model, we use the standard *filtered hits@10* and *filtered mean reciprocal rank* metrics. The final model selected after training is that which maximizes the estimated mean reciprocal rank for the validation dataset.

As EmbedS allows for a geometrical interpretation of triples, we also evaluate a triple classification performance. For each triple in the dataset, and an equal amount of randomly generated false triples (i.e. triples not in the dataset), if $t = (e_i, p_k, e_j)$, the binary classification consists in asking whether the $2n$ -dimensional point (e_i, e_j) is contained in the sphere centered at (p_k^α, p_k^β) with radius σ_k or not. Precision and recall values are obtained for this test.

EmbedS was trained on the wn18 dataset, optimizing for best validation (filtered) hits@10 value, obtaining 94.9%, which is comparable to state-of-the-art models such as HoLE [10]. HoLE is clearly superior in the mean reciprocal rank metric, however, with a value of 0.938, compared to 0.560 for EmbedS. It must be noted, though, that these values correspond to *harmonic* mean ranks of 1.07 for HoLE and 1.79 for EmbedS. If EmbedS is now instructed to optimise the geometrical interpretation, we achieve a precision of 84.2% and a recall of 83.9%, corresponding to an f-measure of 84.0%.

On the dbpedia_v2 dataset, we find that EmbedS achieves a performance on hits@10 of 22.7% and a mean reciprocal rank of 0.133 (corresponding to a harmonic mean rank of 7.52). TransE, on the other hand, performs at 11.6% hits@10 and 0.054 mean reciprocal rank (corresponding to a harmonic mean rank of 18.52).

5 CONCLUSIONS

In this paper we study the new problem of training graph embeddings on ontology-rich datasets. We propose a model which considers RDFS classes and other ontological information as first-class citizens, providing a geometrical interpretation for triples and for ontology assertions. Preliminary experimental results show that the model can perform at state-of-the-art levels on standard benchmark datasets, while on ontology-rich datasets it is also able to provide an alternative form of triple classification which takes advantage of the geometrical interpretation. An exhaustive experimental evaluation is required to be able to fully understand the limitations of this new model, although the encouraging results shown seem to indicate that incorporating ontological information into graph embedding models can potentially open a new avenue of research.

REFERENCES

- [1] M. Arenas, C. Gutierrez, and J. Pérez. Foundations of RDF databases. In *Reasoning Web'09*.
- [2] J. Bergstra and Y. Bengio. Random Search for Hyper-Parameter Optimization. *JMLR'12*.
- [3] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Durán, Jason Weston, and Oksana Yakhnenko. Translating Embeddings for Modeling Multi-relational Data. In *NIPS'13*.
- [4] Kai-Wei Chang, Wen-tau Yih, Bishan Yang, and Christopher Meek. Typed Tensor Decomposition of Knowledge Bases for Relation Extraction. In *EMNLP'14*.
- [5] L. A. Galárraga, C. Teflioudi, K. Hose, and F. M. Suchanek. AMIE: association rule mining under incomplete evidence in ontological knowledge bases. In D. Schwabe, V. A. F. Almeida, H. Glaser, R. A. Baeza-Yates, and S. B. Moon, editors, *22nd International World Wide Web Conference, WWW '13, Rio de Janeiro, Brazil, May 13-17, 2013*, pages 413–422. International World Wide Web Conferences Steering Committee / ACM, 2013.
- [6] Xueyan Jiang, Volker Tresp, Yi Huang, and Maximilian Nickel. Link Prediction in Multi-relational Graphs using Additive Models. In *SeRSy'12*.
- [7] Yankai Lin, Zhiyuan Liu, Maosong Sun, Yang Liu, and Xuan Zhu. Learning Entity and Relation Embeddings for Knowledge Graph Completion. In *AAAI'15*.
- [8] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient Estimation of Word Representations in Vector Space. *CoRR'13*.
- [9] Maximilian Nickel, Kevin Murphy, Volker Tresp, and Evgeniy Gabrilovich. A Review of Relational Machine Learning for Knowledge Graphs. *Proceedings of the IEEE'16*.
- [10] Maximilian Nickel, Lorenzo Rosasco, and Tomaso A. Poggio. Holographic Embeddings of Knowledge Graphs. In *AAAI'16*.
- [11] Maximilian Nickel, Volker Tresp, and Hans-Peter Kriegel. A Three-Way Model for Collective Learning on Multi-Relational Data. In *ICML'11*.
- [12] M. Sadoghi, K. Srinivas, O. Hassanzadeh, Y. Chang, M. Canim, A. Fokoue, and Y. A. Feldman. Self-curating databases. In *EDBT'16*.
- [13] Zhen Wang, Jianwen Zhang, Jianlin Feng, and Zheng Chen. Knowledge Graph Embedding by Translating on Hyperplanes. In *AAAI'14*.

All that Incremental is not Efficient: Towards Recomputation Based Complex Event Processing for Expensive Queries

Abderrahmen Kammoun¹, Syed Gillani², Julien Subercaze¹, Stephane Frenot², Kamal Singh¹,
Frédérique Laforest¹ and Jacques Fayolle¹

¹UJM-Saint-Etienne, CNRS, Laboratoire Hubert Curien, Saint-Etienne, France

²Univ Lyon, INSA Lyon, Inria, CITI, F-69621 Villeurbanne, France

{firstname.lastname}@univ-st-etienne.fr¹, {firstname.lastname}@insa-lyon.fr²

ABSTRACT

Complex Event Processing (CEP) deals with matching a stream of events with the query patterns to extract complex matches. These matches incrementally emerge over time while the partial matches accumulate in the memory. The number of partial matches for expressive CEP queries can be polynomial or exponential to the number of events within a time window. Hence, traditional strategies result in an extensive memory and CPU utilisation. In this paper, we revisit the CEP problem through the lens of complex queries with expressive operators (*skip-till-any-match* and *Kleene+*). Our main result is that traditional approaches, based on the partial matches' storage, are inefficient for these types of queries. We advise a simple yet efficient recomputation-based technique that experimentally outperforms traditional approaches on both CPU and memory usage.

1 INTRODUCTION

Complex Event Processing (CEP) matches a sequence of events within a stream against a complex query pattern that specifies constraints on the extent, order, values, and quantification of the matching events. Most of the CEP systems incrementally produce matched patterns, where partial matches are stored and then computed to avoid the recomputation cost [11, 14]. That is, with the arrival of an event, a CEP system (i) can generate a new partial match by matching incoming event with the prefix of the defined query pattern; (ii) checks with the existing partial matches if the incoming event can be part of them or complete them. The number of partial matches for such strategy can be polynomial or exponential to the number of events within a window [1, 9, 14]: some partial matches lead to the complete matches while others fail. This results in extensive memory and CPU utilisations. In the following, we present a real-world CEP query to showcase the issues of incrementally processing partial matches.

Example 1. A stock market application processes thousands of financial transactions per second to detect patterns that signify emerging profit opportunities. An example of such a pattern, called V-shaped pattern [12, 14], is described in Query 1 using the syntax from SASE [13]. Query 1 detects an increasing and then decreasing pattern per company. Hence, the price of the events must first increase from an initial value ($a.price < b.price$), then the events should show an uptrend ($b.price < NEXT(b.price)$) using the *Kleene+* operator, and then the price of the matched events should decrease such that it is less than the first reported increase ($c.price < FIRST(b.price)$). All the matched events should

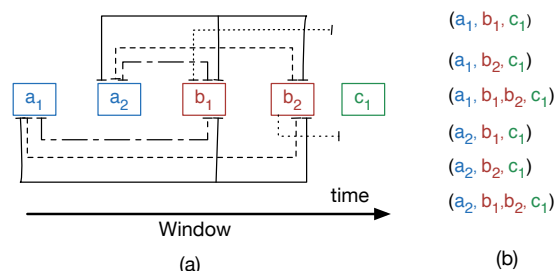


Figure 1: (a) Partial matches for Query 1 over the event stream, (b) complete matches for the Query 1

be within a window of size 30 minutes which slides every 2 minutes.

```

QUERY 1. PATTERN SEQ (a, b+, c) WHERE [companyID]
AND a.price < b.price AND
b.price < NEXT(b.price) AND c.price < FIRST(b.price)
WITHIN 30 minutes SLIDE 2 minutes

```

To reveal all the profit opportunities, Query 1 detects all the combinations of patterns using an operator called as *skip-till-any-match* event selection strategy in the literature [1, 12, 14]. This operator assists in ignoring the local price fluctuations to preserve opportunities for detecting longer and thus more reliable patterns. Fig. 1 shows the evaluation of Query 1 over an event stream. From Fig. 1 (a), before the arrival of an event c_1 , six partial matches (shown with the connected lines) for a_1 and a_2 are produced, and one for each b_1 and b_2 . Note that the prefix of the Query 1 is highly unselective ($a.price < b.price$) and any event can start a partial match. Consequently, there can be a large number of partial matches that would not produce full matches, which result in wasted computation. The complete set of matches, after the arrival of a *trigger* event c_1 are shown in Fig. 1 (b).

State of the art and limitations. The issues with a large number of partial matches and their effect on the memory and CPU resources have been acknowledged in the literature [1, 9, 12, 14]. The usual remedy proposed for such issues is to factorise the commonalities between the partial matches that originate from the same set of events [12, 14]. Hence, the query evaluation is broken into two phases. The first phase tracks the commonalities between the partial matches and compresses them using an additional data structure, e.g. an events graph. The second phase constructs the complete matches while decompressing the set of common partial matches, e.g. through *depth-first-search*. This strategy can reduce the memory requirements from exponential to polynomial, at the cost of the compression/decompression operations. Moreover, since a sub-partial match can be part of

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

multiple complete matches, this strategy recomputes common sub-partial matches for each match that contains one. This results in redundant computations, and the high cost of maintaining additional structure and reconstruction of the matches remains quite significant. For instance, in Fig.1 (a), all the partial matches are kept – and computed – even if a trigger event, i.e. c_1 , never arrives within a window. Finally, due to the high space complexity, these systems spend a considerable amount of time in repeated memory allocation and reclamation with the expiration of a window.

Contributions. This paper initiates the study of a recomputation-based CEP that addresses the following two main points.

- Since storing partial matches is expensive, only events should be stored in a window. This reduces the memory cost from polynomial or exponential to linear
- The matches should be recomputed only when the trigger events’ arrive and results should be directly stored on the disk. Hence, redundant computations are not performed and only the subsets of events are processed to produce the matches. Furthermore, since matches are directly stored on the disk, repeated in-memory allocation and reclamation operations are avoided.

In theory, both recomputation and incremental evaluation techniques have the same worst-case run-time cost. However, in practice, recomputation process provides the following properties: (1) it is performed for a fewer number of times (depending on the trigger events); (2) it alleviates the redundant computations that arrive due to the partial matches that would not produce a complete match; (3) it avoids the cost of creation and deletion of partial matches.

The detailed structure and contributions of this paper are as follow. We first provide the compilation of the query tree for a CEP query (§ 2.2). We then present the design of an algorithm for recomputing matches and analyse its complexity (§ 2.3). Based on this algorithm, we present the techniques to process joins between events and to execute the Kleene+ operator (§ 2.4). We implemented our solution and demonstrate that it outperforms existing solutions both in terms of memory and CPU cost (§ 3).

2 RECOMPUTATION BASED CEP

2.1 Preliminaries

In this section, we present the CEP specific definitions, query representation and query evaluation techniques.

Event. An event e is tuple (A, t) , where $A = \{A_1, A_2, \dots, A_m\}$ ($m \geq 1$) is a set of attributes and $t \in \mathbb{T}$ is an associated timestamps that belongs to a totally ordered set of time points (\mathbb{T}, \leq) .

Event Stream. An event stream \mathcal{S} is a possibly infinite set of events such that for any given timestamps t and t' , there is a finite amount of events occurring between them.

Event Sequence. A chronological ordered sequence of events, with a total ordering given by \mathbb{T} is represented as $\vec{E} = \langle e_1, e_2, \dots, e_n \rangle$ with e_1 refer to the *first* event and e_n to the *last*.

CEP Query. A CEP query Q has the following form:

PATTERN P [WHERE Θ] WITHIN w SLIDE s

where $P = \text{SEQ}(p_1, \dots, p_k)$ is a sequence of pairwise disjoint variables of the form p and p^+ , $\Theta = \theta_1, \dots, \theta_l$ is a set of predicates (constant and variable) over the variables in P (see Query 1 in Example 1), w is the time window and s is slide of the window to define the scope of event stream. A variable $p \in P$ binds a sequence of a single event $\langle e_i \rangle$, while the qualified variables

$p^+ \in P$ binds a sequence of one or more events $\langle e_1, \dots, e_n \rangle$, $n \geq 1$ for a query match.

CEP Query Match. To define the matching of a CEP query Q , we use a substitution $\gamma = \{p_1/\vec{E}_1, \dots, p_k/\vec{E}_k\}$ to bind the event sequences (\vec{E}) with the variables. Given Q and the event stream \mathcal{S} , a substitution γ is a match of Q in \mathcal{S} , iff (i) all the predicates Θ in Q evaluate to true, (ii) for events in the two event sequences $e \in \vec{E}_i$ and $e' \in \vec{E}_{i+1}$, we have $e.t < e'.t$ and (iii) all the events in each event sequence \vec{E}_i has timestamps less than the defined window w . Since no order is imposed on the selected events, it complies to the skip-till-any-match selection strategy.

2.2 Query Tree

Given the CEP query, we need to compile it from the high-level language into some form of automaton [1, 4, 6, 13] or a tree-like [5, 11] structure to package the semantics and execution framework. Since we are working with the recomputation-based model, a traditional tree structure customised for the streaming and recomputation settings would suit our needs. Given Q we construct a tree, where leaf nodes are the substitution pairs, i.e. (p_i/\vec{E}_i) , to store the primitive events and the internal nodes represent the joins on the defined predicates Θ and temporal ordering. We call it a query tree \mathcal{T}_q . Our model differs from other tree-structures [5, 11] since we do not store any partial matches. An example of such a tree for Query 1 is shown in Fig 2, where we have three leaf nodes for the variables bindings a/\vec{E}_1 , b/\vec{E}_2 and c/\vec{E}_3 . The internal nodes in Fig. 2 evaluate the defined Θ in terms of joins (denoted as \bowtie_{Θ}) for all the variables $p \in P$. Furthermore, since for a CEP query, the matched events should follow the sequential order, joins on the timestamps (denoted as \bowtie^t) are also provided in the \mathcal{T}_q .

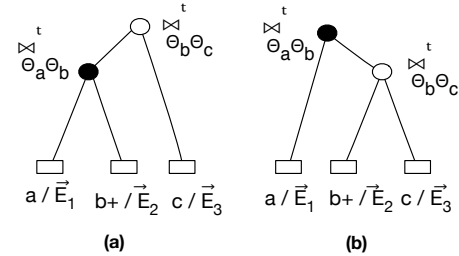


Figure 2: (a) Left-deep and (b) Right-deep Query tree for Query 1 in Example 1

2.3 Query Evaluation

We now present the algorithm to evaluate the query tree over the event stream without storing the partial matches. Algorithm 1 shows the query evaluation and is divided into three main steps.

Step 1. For each incoming event e , we add e to the compatible event sequence \vec{E}_i , such that constant predicates (e.g. $a.price > 10$) filter the unwanted events for each \vec{E}_i in \mathcal{T}_q . For instance, for a constant predicate $a.price > 10$, all the events in a/\vec{E}_1 , should have the $price > 10$. This step (lines 4-6) constitutes to the accumulation of events within a defined window.

Step 2. For each incoming event e , check if it can trigger the query evaluation to produce matches. That is, if e can be part of \vec{E}_k ($k = |P|$), it can complete a set of matches; since it contains the highest timestamp within the window. For instance, Query 1

Algorithm 1: CEP Query Evaluation

Input: Query Tree \mathcal{T}_q and an event stream \mathcal{S}
Output: A set of query matches

```

1  $Q \leftarrow (P, \Theta, \omega, s);$  // CEP Query
2  $\vec{E} \leftarrow \{\vec{E}_1, \vec{E}_2, \dots, \vec{E}_k\}, k = |P|;$  // Event sequences for  $\mathcal{T}_q$ 
3 for each  $e \in \mathcal{S}$  do
4   for each  $\vec{E}_i \in \vec{E}$  do
5     if  $\text{isCompatible}(\vec{E}_i, e)$  then
6        $\vec{E}_i = \vec{E}_i \cup e;$  // Step 1
7   if  $\text{isCompatible}(\vec{E}, e)$  then
8      $\text{ExecuteJoins}(\vec{E}, \Theta);$  // Step 2
9      $\text{ExecuteKleenePlus}(\vec{E}, \Theta);$  // Step 3
```

produces the matches only when an event of type c arrives. Hence, we execute the query tree for a trigger event. By execution, we mean executing the joins between events within each \vec{E}_i using the predicates Θ and timestamps t . This step (lines 7-8) assembles all the events, in a batch manner, for each \vec{E}_i that can produce the set of matches.

Step 3. For a $p_i^+ \in P$, we need to compute all the combinations for the events in p_i^+ / \vec{E}_i , i.e. a power set of events in \vec{E}_i . For instance, in Fig. 1 and using Query 1, each a event has $2^2 - 1$ matches for two b events. This step (line 9) groups all the combinations by following the one or more semantics of the Kleene+ operator.

2.4 Detailed Analysis

We now present the details of the two main processes of Algorithm 1, i.e. joining the set of events and computing the power set of events for the Kleene+ operator.

Execution of Joins. Let \vec{E}_i and \vec{E}_j are two event sequences with theta-join $\vec{E}_i \bowtie_{\Theta}^t \vec{E}_j$ over the timestamp t and predicates Θ . Hence, we have joins on multiple relations for the **Step 2**. The generic cost of such joins, i.e. pairwise join, is $O(|\vec{E}_i||\vec{E}_j|)$ and the problem of its efficient evaluation resembles the traditional theta-joins with inequality predicates [8]. The wide range of methods for this problem includes: the textbook merge-sort, hash-based, band-join and various indices such as Bitmap [7]. These techniques are mostly focused on equality joins using a single join relation, however. The inequality joins on multiple join relations are notoriously slow and multi-pass *projection-based* strategies [3, 8] are usually employed. These strategies, however, require multiple sorting operations, each for a distinct relation, and are only optimised for the static datasets, where indexing time is not of much importance. Considering this, we employ the general nested-loop join for our preliminary algorithm. Our experimental analysis showcase that even such a naive algorithm provides competitive performance.

Execution of Kleene+ Operator. For **Step 3**, we need to create all the possible combinations of matches over the joined events. That is, enumerating the powerset of event sequences' with p^+ bindings. A traditional solution in this context would be to generate Gray code sequence of events with p^+ bindings, where a new match can be constructed from its immediate predecessor by adding or removing an event. However, this would require storing the predecessor matches to produce the next one and would result in an extra load on the memory resources. To implement Kleene+ operator efficiently, we use the joined results (from **Step 2**) while generating the binary representation of the

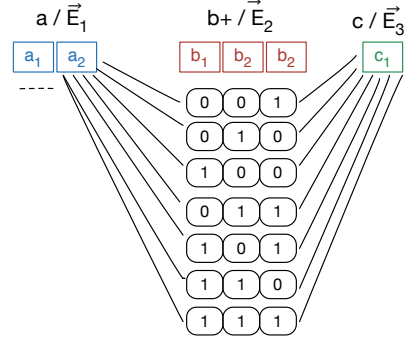


Figure 3: Execution of the Kleene+ operator using the Banker's sequence and generated binary numbers

possible matches using the Banker's sequence [10]. That is, we check the number of events in event sequences' with p^+ bindings after the join process. For $|m|$ number of such events, we need to create 1 to $2^m - 1$ matches. This means if we generate all binary numbers from 1 to $2^{|m|} - 1$, and translate the binary representation of numbers according to the location of the events in the p^+ / \vec{E}_i , we can produce all the matches for the Kleene+ operator in a batch manner. For instance, consider Fig. 3 (using Query 1), where there are three b events for the Kleene+ operator. Hence, we generate binary numbers from 1 to $2^3 - 1$. Now equates 1 as take element at the specified location of the \vec{E}_2 and 0 as do not take the element. Then using the generated binary numbers, we generate all the combinations of matched events.

Complexity Analysis. Herein, we briefly present the complexity analysis for the three steps described in Algorithm 1. **Step 1** results in a constant time operation since an incoming event can be directly added to an event sequence. **Step 2** has a polynomial time-cost (pair-wise joins) and depends on the number of patterns P defined in a CEP query. For n events in a window and $k = |P|$, we have $O(n^k)$. **Step 3** requires producing an exponential number of matches for a Kleene+ operator. For n events in a window, we have $O(2^n)$. The memory cost for the Algorithm 1 is linear to the number of events within a window.

3 EXPERIMENTAL EVALUATION

In this section, we report the results of our experimental study on both incremental and recomputation-based methods for CEP. Our proposed techniques have been implemented in Java and our system is called RCEP. All the experiments were performed on a machine equipped with Intel Xeon E3 1246v3 processor and 32 GB of memory. For robustness, each experiment was performed 3 times and we report median values.

Datasets. We employ both real and synthetic datasets to compare the performance of our proposed techniques.

Synthetic Stock Dataset (S-SD): We use the SASE++ generator, as used in [14], to produce the synthetic dataset. Each event carries a timestamp, company-id, volume and the price of a stock. This dataset enables us to tweak the selectivity measures of matches $\frac{\#ofMatches}{\#ofEvents}$ to evaluate the performance of the systems at different workloads. In total, the generated dataset contains 1 million events.

Real Credit Card Dataset (R-CCD): We use a real dataset of credit card transactions [2]. Each event is a transaction accompanied by several arguments, such as the time of the transaction,

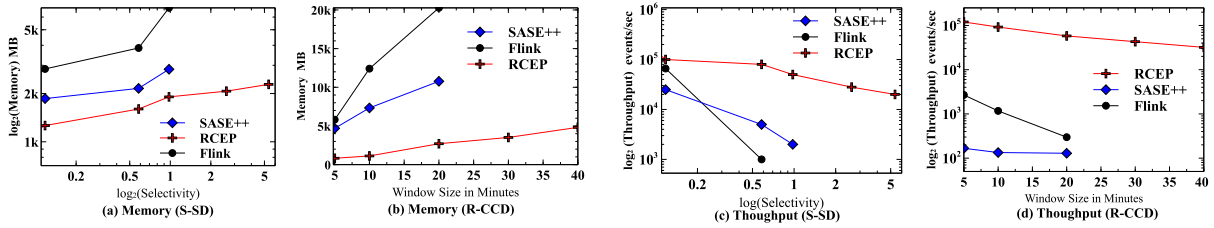


Figure 4: Memory cost and throughput analysis of the CEP systems over the two datasets S-SD and R-CCD

the card ID, the amount of money spent, etc. This dataset contains around 1.5 million events.

Queries. We consider 8 different variations of Query 1 (Section 1) against the stock dataset. These queries variations differ by the constant predicates and time window to control the selectivity of produced matches. For the credit card dataset, we use a CEP query describing “Big after Small” pattern [2] using the SEQ($a, b+$) template. That is, an outstandingly large amount of transactions after one or a series of small amounts.

Methodology. We compare RCEP with the SASE++ [14] and the open source streaming system Apache Flink [6]. All of these systems support skip-till-any-match and Kleene+ operator: both SASE++ and Flink employ incremental evaluation of partial matches. Flink guarantees that events are processed in parallel but in-order by their timestamps. Unless otherwise specified, all experiments use a slide granularity $s = 1$. We measure two standard metrics common for the CEP systems: throughput and memory requirements [1, 12, 14]. The memory requirements were measured by considering the resident set size (RSS) in MBs. RSS was measured using a separate process that polls the /proc Linux file system, once a second. We use the selectivity measures $\frac{\#of\ Matches}{\#of\ Events}$ and window size to test different workloads.

Memory Cost. Figs. 4 (a) and (b) show the memory consumption of all the systems for both datasets. As expected, increase in the selectivity measures (subsequently window sizes) results in a large number of partial matches and an extensive memory cost for both SASE++ and Flink. In particular, Apache Flink consumption is exponential in terms of the number of events in the window. SASE++ managed to sustain memory requirement due to the superior compression of Kleene+ matches. However, with the increase in the number of events that prefixed a new partial match, its memory utilisation increases to about two orders of magnitude compared to our recomputation-based approach. This phenomenon is largely observed in Fig. 4 (b), where the credit card dataset contains a large number of events that can initiate a new partial match. In contrast, RCEP scales linearly to the number of events within a window and not the partial matches.

CPU Cost. Figs. 4 (c) and (d) show the relative performance of the CEP systems over both datasets. We can see that, in general, RCEP have much higher throughput (more than an order of magnitude) than Flink and SASE++. As a matter of fact, SASE++ and Flink do not produce results for several hours for the moderate selectivities and window sizes. This is because the cost of SASE++ and Flink is highly dependent on the number of partial matches within a window. As the window size (subsequently selectivity) increases, both systems produce a large number of partial matches and spend most of their time in compressing and decompressing of the common events within the partial matches. That is, traversing through the stack of pointers using depth-first-search to extract

all the matches. In contrast, (i) RCEP initiates the recomputation of matches only if the triggered events’ arrive; (ii) the execute joins over the stored events; and (iii) the Kleene+ operator is executed only for the events that can be part of the final matches. Hence, RCEP performs much better and consume less memory than SASE++ and Flink, often by 1-2 orders of magnitude.

4 LOOKING AHEAD

In this preliminary study, we have highlighted the utility of recomputation-based CEP for expensive CEP queries. We have proposed our first algorithm for recomputing the matches with the arrival of new events. To our knowledge, ours is the first algorithm of this kind in the context of CEP. Our experimental results show that recomputation-based approach outperforms the incremental approach used by the existing systems.

Our study opens up several directions for the future work. A major direction is to establish techniques to efficiently store and index events within a defined window. Without this, we cannot discard events within an event sequence unless it is accessed and compared with all the other events. Hence, the indexing of events would enable us to prune irrelevant events before the joining process. Further, we plan to consider new algorithms for the multi-relational and inequality joins in the streaming settings since existing algorithms are only effective for the static workloads and require extensive indexing time. Finally, we would like to incorporate our solution in the open-source Apache Flink framework.

REFERENCES

- [1] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *SIGMOD*, pages 147–160, 2008.
- [2] A. Artikis, N. Katzouris, I. Correia, C. Baber, N. Morar, I. Skarbovsky, F. Fournier, and G. Paliouras. A prototype for credit card fraud management: Industry paper. In *DEBS*, pages 249–260, 2017.
- [3] J. L. Bentley and J. H. Friedman. Data structures for range searching. *ACM Comput. Surv.*, 11(4):397–409, Dec. 1979.
- [4] L. Brenna, A. Demers, J. Gehrke, M. Hong, J. Ossher, B. Panda, M. Riedewald, M. Thatte, and W. White. Cayuga: A high-performance event processing engine. In *SIGMOD*, pages 1100–1102, 2007.
- [5] ESPER. <http://www.espertech.com/esper/>.
- [6] A. Flink. <https://flink.apache.org/>.
- [7] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database systems - the complete book (international edition)*. Pearson Education, 2002.
- [8] Z. Khayyat, W. Lucia, M. Singh, M. Ouzzani, J.-A. Papotti, N. Tang, and P. Kalnis. Fast and scalable inequality joins. *The VLDB Journal*, pages 125–150, 2017.
- [9] I. Kolchinsky, I. Sharfman, and A. Schuster. Lazy evaluation methods for detecting complex events. In *DEBS*, pages 34–45, 2015.
- [10] J. Loughry, J. van Hemert, and L. Schoofs. Efficiently enumerating the subsets of a set. *Applied-math*, 2000.
- [11] Y. Mei and S. Madden. Zstream: A cost-based query processor for adaptively detecting composite events. In *SIGMOD*, pages 193–206, 2009.
- [12] O. Poppe, C. Lei, S. Ahmed, and E. A. Rundensteiner. Complete event trend detection in high-rate event streams. In *SIGMOD*, pages 109–124, 2017.
- [13] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *SIDMOD*, 2006.
- [14] H. Zhang, Y. Diao, and N. Immerman. On complexity and optimization of expensive queries in complex event processing. In *SIGMOD*, 2014.

DeepEye: Visualizing Your Data by Keyword Search

[Visionary Paper]

Xuedi Qin

Department of Computer Science, Tsinghua University
qxd17@mails.tsinghua.edu.cn

Nan Tang

Qatar Computing Research Institute, HBKU
ntang@hbku.edu.qa

Yuyu Luo

Department of Computer Science, Tsinghua University
luoyuyu@mail.tsinghua.edu.cn

Guoliang Li

Department of Computer Science, Tsinghua University
liguoliang@tsinghua.edu.cn

ABSTRACT

Do you dream to create good visualizations for your dataset simply like a Google search? If yes, our visionary system DEEPEYE is committed to fulfill this task. Given a dataset and a keyword query, DEEPEYE understands the query intent, generates and ranks good visualizations. The user can pick the one he likes and do a further faceted search to easily navigate the visualizations. We detail the architecture of DEEPEYE, key components, as well as research challenges and opportunities.

1 INTRODUCTION

Nowadays, the ability to create good visualizations has shifted from a nice-to-have skill to a must-have skill for all data analysts. However, the overwhelming choices of interactive data visualization tools (e.g., Tableau, Microsoft Excel and D3 [5]) only allow experts to create good visualizations, assuming that the experts know many details: the meaning and the distribution of the data, the right combination of attributes, and the right type of charts – these requirements are apparently not easy, even for experts.

Unfortunately, creating good data visualization is hard. From the user perspective, there are many possible ways of visualizations for a given dataset (for example, different attribute combinations and visualization types), and many ways of transforming data (for example, grouping, binning, sorting, and a combination thereof) – these make it infeasible for the user to enumerate all possible visualizations and select the ones he needs. From the system perspective, among numerous problems, no consensus has emerged to quantify the “goodness” of a visualization. What makes it harder is when the system does not even know what the user wants. Recently, there have been proposals for visualization recommendation systems [15, 18], which focus on automatically recommending “interesting” visualizations from a diversified criteria, such as relevance, surprise, non-obviousness, diversity and coverage. However, as pointed out by [3], these systems may mislead the user, by generating visualizations that might be *worse than nothing*, since it is basically impossible to guess a user’s query intent from nothing.

The natural problem arises: *What is the most feasible way to (automatically) create good visualizations even for dummies?*

We have three key intuitions for handling the above problem.

(1) *What is the ideal language for the user to specify his intent?* Of course, the mother tongue – Google-like natural language search interface is friendly to everyone. Note that (i) most visualizations

are generated using declarative languages like Vega-Lite [12]; and (ii) using machine learning to convert text to SQL queries for relational databases has been proved effective [11]. Hence, we can allow users to pose keyword queries and our system automatically converts them to declarative visualization queries.

Keyword-to-Visualizations is hard: keywords are typically ambiguous and underspecified; and good visualizations concern statistical properties such as trends, comparisons, which are rather hard to grasp even for human.

(2) *How can we solve the fundamental cognitive science problem for quantifying good visualizations?* Evidently, good visualizations exist, which can be collected from many sources that take experts hours or even days to produce such valuable visualizations. Naturally, the research problem is how to transfer the knowledge from these known good visualizations to judge a new visualization.

Transferring good visualizations. The basic intuition is: a new visualization is good, if it *matches* some known good visualization (see Section 2.2 for a further discussion).

(3) *How can we rank visualizations?* Ranking is the secret sauce to any search engine. Note, however, that it is almost impossible to rank visualizations, if they are independent of each other.

Visualization link graph. We propose a novel graphical model to link good visualizations. Informally speaking, two visualizations are linked if they are *relevant*, which are further classified into different types (or facets), e.g., similar or diverse that are captured by different facet functions (or criteria [15]).

The main benefit of having the above graph is twofold. (i) We can devise *PageRank* [6] like algorithms to rank visualizations. (ii) We can leverage the edge types to provide faceted search (*a.k.a.* faceted browsing) such that the user can easily set the compass to navigate the ocean of visualizations by simple clicks.

DEEPEYE is our visionary system to create good visualizations by *keyword search and simple clicks*.

DEEPEYE Workflow. A user can pose a keyword query K . DEEPEYE translates this keyword query to multiple candidate visualizations V , discovers good visualizations V' in V , ranks V' , and returns the top ones V'' . When the user selects a visualization V in V'' and further explores by clicking a facet of V , DEEPEYE discovers more visualizations and helps the user to easily explore his desired visualizations. The user may iterate over the above processes until he finds all visualizations that he wants.

Remarks. (1) Different from *visualization recommendation systems* [15, 18], (i) DEEPEYE is a visualization search engine that the user needs to provide his intent – we do not believe in the magic that one can guess something from nothing; and (ii) DEEPEYE decides and ranks good visualizations by comparing with

*Guoliang Li is the Corresponding Author.

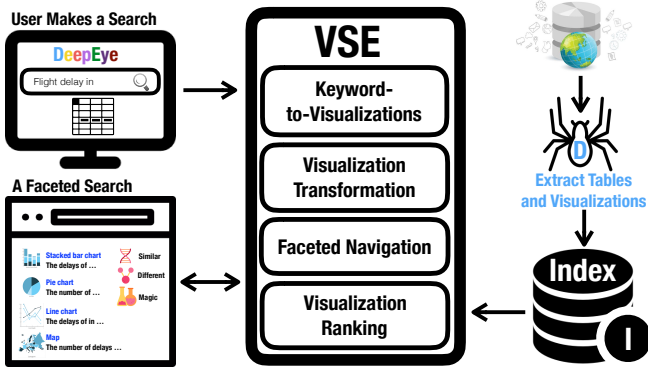


Figure 1: The Architecture of DEEP EYE

known good visualizations. (2) After finding good visualizations, DEEP EYE will support to export them to a designated interactive data visualization tool (e.g., Tableau) for more customized manipulations. (3) Although intuitively, DEEP EYE, as a search engine, shares a similar architecture with well known search engines such as Google and Bing, there are many new challenges for designing visualization search algorithms (see Section 3).

2 THE DEEP EYE ARCHITECTURE

The architecture of DEEP EYE is given in Figure 1. DEEP EYE crawls, stores and indexes good visualizations from multiple sources.

The user starts by posing a *keyword search* and provides a dataset. The *visualization search engine* (VSE) will first generate a set of visualizations V by the *Keyword-to-Visualizations* module. These candidate visualizations V will be matched with known good visualizations by the *Visualization Transformation* module that produces $V' \subseteq V$, which will then be ranked by the *Visualization Ranking* module and the top ones $V'' \subseteq V'$ are returned to the user. The user may pick the one he likes and discovers more visualizations by the *Faceted Navigation* module. This module aims to reduce the number of user interactions and helps users to find the target visualizations as soon as possible.

2.1 Preliminary

Datasets. We crawl visualizations with data and visualizations charts (e.g., pie/line/bar charts) from multiple sources. Then we use them to visualize a user-given dataset D . For simplicity, we consider D as only one table, which can be easily extended to support multiple tables by relational joins (see SeeDB [16]).

Data Features. Typically, what will decide whether a visualization of a dataset is good or not depends more on its features (or representations [2]), not its data values. More specifically, we consider the following features of a dataset D : the data type of a column (e.g., categorical, numerical, and temporal), the number of distinct values of a column, the number of tuples in a column, the ratio of unique values in a column, the $\max()$ and $\min()$ values of a column, and statistical correlation between two columns (e.g., linear, polynomial, power, and log).

Visualization Queries. For *declarative visualization language*, we use Vega-Lite [12], a high-level grammar that enables rapid specification of interactive data visualizations. (Note that our system can support any declarative visualization query language.) Each *query* Q , specified in the Vega-Lite JSON format over the dataset D , denoted by $Q(D)$, will produce a visualization. A sample Vega-Lite query is as follows:

Edge labels: s (similar), d (diverse), c (coverage)

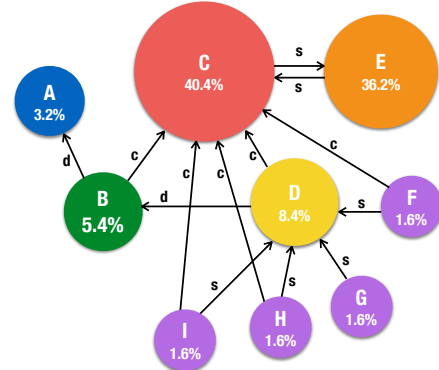


Figure 2: A Sample Visualization Link Graph

```

{
  "data": {"url": "flights.json"},
  "mark": "bar",
  "encoding": {
    "x": {
      "bin": true,
      "field": "carriers",
      "type": "qualitative"
    },
    "y": {
      "aggregate": "count",
      "type": "quantitative"
    }
  }
}

```

Visualization Crawler. The DEEP EYE *crawler* will extract tables and their associated visualizations from multiple sources, where *both* the table *and* the visualization specifications (or equivalent queries) need to be explicitly given. For example, there are hundreds of visualization examples in <https://www.highcharts.com>, with both data values and visualization specifications. Note that we do not require these sources to use Vega-lite – most visualization specifications can be easily converted to Vega-lite queries, for which we need to implement corresponding tools for query rewriting. When only the data and visualizations are given but the visualization queries are absent, an interesting open problem is to automatically infer the declarative visualization queries.

Visualization Link Graph. A *visualization link graph* is a directed graph $G(N, E)$ with nodes N and directed edges E . Each node $v \in N$ is a visualization (i.e., $Q(D)$). Each directed edge $e : (u, v) \in E$ can have multiple labels, denoted by $L(e)$, where each label $L(e) \in L(e)$ is a facet, such as similar, diverse, coverage.

In order to decide the edges and their labels, we have defined a set of *facet functions*. For example, there is a “similar”-edge from node u to node v , if the corresponding *similar-function* $f_s(u, v)$ returns *true*, denoting that u is similar to v . There can have another “similar”-edge from v to u if $f_s(v, u)$ also returns *true*. Note that any *facet function* does not have to be symmetric – $f(u, v)$ is *true* does not imply that $f(v, u)$ is *true* as well. The other edges and labels are generated similarly using different *facet functions*. Please find more details in the *Faceted Navigation* module in Section 2.2.

A sample visualization link graph is given in Figure 2. The number associated with each node denotes the importance of the node, which will be further discussed in Section 3.

2.2 Visualization Search Engine Modules

In this section, we will discuss the functionalities and our basic designs for the modules of DEEP EYE visualization search engine.

Keyword-to-Visualizations. Given a keyword query K and a dataset D , this module generates all candidate visualizations.

As mentioned earlier, this problem is hard because keyword queries are always ambiguous and underspecified – there might have a large number of candidate visualizations due to different attribute combinations and multiple data transformation operations (e.g., grouping, binning, sorting). Fortunately, there are simple observations about good/bad visualizations, e.g., pie charts are best to use when comparing parts of a whole, and they do not show changes over time; and bar graphs are used to compare things between different groups or to track changes over time, and too many bars (e.g., > 50) are hard for human to interpret. The traditional wisdom from visualization experts can be encoded into rules to prune many apparently bad charts (see e.g., <https://www.pinterest.com/pin/20125529565819990/> for a chart type cheat sheet that can be easily leveraged).

Visualization Transformation. The broad intuition is that, if a visualization V can “match” a known (crawled) good visualization, then V is probably good. As mentioned earlier in the data features, visualization matching focuses more on feature matching (e.g., similar domains and similar trends), in contrast to traditional value matching of strings.

Given a visualization V and another visualization N , it is to compute the similarity between their features, which typically falls in the range $[0, 1]$; and we say that V matches N if their similarity is above a threshold σ .

Given a set of visualizations V , the module of *Visualization Transformation* will find a set of existing good visualizations N that match these candidate visualizations. Note that one candidate query can match multiple good visualizations, and vice versa. Also, a candidate visualization that cannot match any existing good one will be removed, which will result in a subset V' of V .

Visualization Ranking. Given a set of visualizations V' and their matched good visualizations N , the *Visualization Ranking* module will rank V' based on N (i.e., a subgraph of the visualization linkage graph) and return to the user. We can either use the learning-to-rank [7] techniques to learn the features from known good visualizations or design new ranking functions. We can also use the user click-through data to rank the visualizations.

Faceted Navigation. When a user picks a visualization V he likes, he can further explore other visualizations by facets. The *Faceted Navigation* module will discover another set of visualizations based on V , which will also be ranked and returned to the user, in order to help users navigate the visualizations.

We plan to implement this module by providing each facet a programming interface (i.e., an API) that implements a *facet function*, e.g., similar, diverse, coverage. Also, we make it extensible by allowing domain experts to plug in other APIs for different facets. The main reason to allow this flexibility is that till now, there is no consensus about criteria of finding interesting visualizations, which remains an open problem. Note that these facet-functions do not need to be mutually exclusive.

3 RESEARCH OPPORTUNITIES

3.1 Visualization Data

An effective visualization system relies on high-quality and high-coverage visualization data. Although there are some open websites that we can crawl some data, the coverage is limited and we need to construct a visualization benchmark.

Opportunity. First, our system requires visualizations with data, visualization queries and visualization charts. Many websites, however, do not contain such data, and we need to infer one dimension based on the others, e.g., inferring a query based on the data and a visualization. Second, we also need some well-ranked visualizations to help us rank the visualizations for a new dataset. However many websites do not contain rankings, and it is challenging to infer rankings. Third, we can use crowdsourcing to collect more visualization data and rank the visualizations, and the challenge is to reduce the crowdsourcing cost, improve the quality, and avoid the redundancy with existing data.

3.2 Visualization Search Engine

3.2.1 Keyword-to-Visualizations. The essential problem is to understand natural language and generate visualization queries. Fortunately, the recent machine learning and deep learning techniques have made it easy to understand natural language (see e.g., the OpenNLP toolkit: <https://opennlp.apache.org>). Furthermore, several approaches have studied the problem of translating natural language to SQL queries, such as NLDBs [1] and NaLIR [11].

Opportunity. Although the above approaches shed some light on our problem, translating natural languages to visualization queries remains a hard problem. The main reason is that when searching visualizations, the user cares more about the statistical properties such as *trends*, *comparisons*, *fast increase*, which are more ambiguous than querying a DBMS such as “return the average number of publications by Bob in each year”. In other words, even if an expert knows precisely what is the meaning of the natural language query of the user, it is still hard for the expert to formulate using the declarative visualization language, since good visualizations are also data dependent.

The research problem is: Given a keyword query K and a dataset D , discovers candidate visualizations $Q(D')$ that the user wants, where D' could be transformed from D (e.g., by operations like binning, grouping, sorting, and a combination thereof).

3.2.2 Visualization Transformation. The fundamental problem of visualization transformation is to transform the knowledge (or features) from known good visualizations to deciding the goodness of unknown visualizations. A possible way is to compute the similarity between two visualizations $Q_1(D_1)$ and $Q_2(D_2)$ (i.e., the known good one). If $Q_1(D_1)$ is very similar to $Q_2(D_2)$ and $Q_2(D_2)$ is good, then by inference, $Q_1(D_1)$ is also good: showing an interesting trend, using the same chart as employed in Q_2 , etc.

Opportunity. The open problem is how to define the similarity function $\text{sim}(Q_1(D_1), Q_2(D_2))$, which relates to many factors.

- (1) The statistical correlation between the two visualizations $Q_1(D_1)$ and $Q_2(D_2)$, such as the same trend.
- (2) The domain similarity, i.e., whether the attributes used for $Q_1(D_1)$ and $Q_2(D_2)$ come from the same domain.
- (3) The type similarity, e.g., whether the data types used in $Q_1(D_1)$ and $Q_2(D_2)$ are both temporal data.

3.2.3 Visualization Ranking. As mentioned earlier, there is still no consensus to quantify the “goodness” of a visualization. Intuitively, it is harder to quantify “better” visualizations.

Opportunity. The great success of search engines has shown us multiple ways of doing link analysis, with the purpose of “measuring” the relative importance within a set of linked webpages. Maybe the most successful story is *PageRank* [6]. We have defined

our visualization link graph (see Figure 2) in a way that we can use a similar idea of PageRank, which is referred to as *VisualRank*. However, implementing *VisualRank* faces four challenges:

- (1) The edges in the visualization link graph and the edges in the Webgraph have different meanings. In the Webgraph, an edge from a page X to a page Y if there exists a hyper-link on page X referring to page Y . In the visualization link graph, the edges have diversified semantics (or facets).
- (2) The search behaviors between a normal search engine and a visualization search engine are quite different.
- (3) We also want to use the graph in faceted navigation to reduce the number of interactions with the user. It is challenging to design multi-goal optimization algorithms.
- (4) Our end goal is not to rank nodes in the visualization link graph, but the unlinked visualizations for an input dataset, for which we need to infer from the “importance values” of their matched good visualizations to rank.

Hence, a good research opportunity is how to design a *VisualRank* algorithm, which shares the basic intuition of PageRank, but serves the purpose of ranking visualizations.

3.2.4 Faceted Navigation. The recent proposal towards visualization recommendation systems [15] made an attempt to define the criteria of recommending good visualizations: relevance, surprise, non-obviousness, diversity and coverage. Another recent work proposes a general-purpose query language for visualization recommendation [18].

Opportunity. We can certainly use these criteria to create our facets. Unfortunately, these are just research hypothesis – whether they can lead to good visualizations is still not well justified. Besides, there is no well accepted implementation for each of them. Hence, it remains open that which facets should be considered and how to implement them.

3.3 Optimization

Response time is critical to real-time applications – response times under one second are usually considered to be good for search engines. Naturally, this requires us to carefully design DEEPEYE, from storage, indexes, to algorithms.

Opportunity. The first opportunity is about storage: What is the best physical representation of the data and the existing visualizations, assuming that they are stored in tables and JSON files? The work RodenStore [8] proposed an adaptive and declarative storage system providing a high-level interface for describing the physical representation of data. The similar idea can be adopted in the problem of storing datasets and visualizations. The second opportunity is how to store the visualization link graph – this will be tightly coupled with all (ranking) algorithms that use this graph. We can also group the visualizations effectively, e.g., in a hierarchy or using partial orders, to help users to quickly find target visualizations. The third challenge is how to design effective index, especially for large datasets. We can borrow the idea from the search engine, e.g., inverted index and forward index. However, it is rather challenging to design indexes for visualization search and ranking, because they involve more data features and more visualization operations.

4 RELATED WORK

Visualization Recommendation Systems. One important line of work is visualization recommendation systems [14–16]. Roughly speaking, given a dataset, they want to automatically

recommend visualizations to the user under different criteria. DEEPEYE differs from them in the following two key aspects: (1) instead of guessing the user’s intent, DEEPEYE accepts keyword search; and (2) DEEPEYE uses existing good visualizations to reason about the input dataset.

Interactive Data Visualization. There are some interactive data visualization systems, such as D3 [5], protovis [4], matplotlib [10], Tableau [17]. We do not plan to reinvent the wheel – DEEPEYE will export the user selected visualizations to the above toolkits for more complicated manipulations.

Visualization Languages. There are many visualization languages, e.g., Vega [13], Vega-Lite [12], VizQL [9], Ermac [19], and DeVIL [20]. We use Vega-Lite as it is simple yet general enough.

5 CONCLUDING REMARKS

With increasing interest and importance of data visualization for data science applications, there is an emerging need for a tool to easily create good visualizations. We believe that DEEPEYE, the first visualization search engine, will lead to interesting and impactful problems for many communities, including: database, information retrieval, machine learning, and visualization.

Acknowledgement. This work was supported by 973 Program of China (2015CB358700), NSF of China (61632016, 61472198, 61521002, 61661166012), and TAL education.

REFERENCES

- [1] I. Androutsopoulos, G. Ritchie, and P. Thanisch. Natural language interfaces to databases – an introduction. *Natural Language Engineering*, 1(1):29–58, 1995.
- [2] Y. Bengio, A. C. Courville, and P. Vincent. Representation learning: A review and new perspectives. *IEEE Trans. Pattern Anal. Mach. Intell.*, 35(8):1798–1828, 2013.
- [3] C. Binnig, L. D. Stefani, T. Kraska, E. Upfal, E. Zraggen, and Z. Zhao. Toward sustainable insights, or why polygamy is bad for you. In *CIDR*, 2017.
- [4] M. Bostock and J. Heer. Protovis: A graphical toolkit for visualization. *IEEE Trans. Vis. Comput. Graph.*, 15(6):1121–1128, 2009.
- [5] M. Bostock, V. Ogievetsky, and J. Heer. D³ data-driven documents. *IEEE Trans. Vis. Comput. Graph.*, 17(12):2301–2309, 2011.
- [6] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Seventh International World-Wide Web Conference (WWW 1998)*, 1998.
- [7] C. J. C. Burges, T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton, and G. N. Hullender. Learning to rank using gradient descent. In *ICML*, pages 89–96, 2005.
- [8] P. Cudré-Mauroux, E. Wu, and S. Madden. The case for rodentstore: An adaptive, declarative storage system. In *CIDR*, 2009.
- [9] P. Hanrahan. Vizql: a language for query, analysis and visualization. In *SIGMOD*, page 721, 2006.
- [10] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science and Engineering*, 9(3):90–95, 2007.
- [11] F. Li and H. V. Jagadish. Understanding natural language queries over relational databases. *SIGMOD Record*, 45(1):6–13, 2016.
- [12] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, and J. Heer. Vega-lite: A grammar of interactive graphics. *IEEE Trans. Vis. Comput. Graph.*, 23(1):341–350, 2017.
- [13] A. Satyanarayan, R. Russell, J. Hoffswell, and J. Heer. Reactive vega: A streaming dataflow architecture for declarative interactive visualization. *IEEE Trans. Vis. Comput. Graph.*, 22(1):659–668, 2016.
- [14] T. Siddiqui, A. Kim, J. Lee, K. Karahalios, and A. G. Parameswaran. Effortless data exploration with zenvisage: An expressive and interactive visual analytics system. *PVLDB*, 10(4):457–468, 2016.
- [15] M. Vartak, S. Huang, T. Siddiqui, S. Madden, and A. G. Parameswaran. Towards visualization recommendation systems. *SIGMOD Record*, 45(4):34–39, 2016.
- [16] M. Vartak, S. Rahman, S. Madden, A. G. Parameswaran, and N. Polyzotis. SEEDB: efficient data-driven visualization recommendations to support visual analytics. *PVLDB*, 8(13):2182–2193, 2015.
- [17] R. M. G. Wesley, M. Eldridge, and P. Terlecki. An analytic data engine for visualization in tableau. In *SIGMOD*, pages 1185–1194, 2011.
- [18] K. Wongsuphasawat, D. Moritz, A. Anand, J. D. Mackinlay, B. Howe, and J. Heer. Towards a general-purpose query language for visualization recommendation. In *HLLDA@SIGMOD*, page 4, 2016.
- [19] E. Wu, L. Battle, and S. R. Madden. The case for data visualization management systems. *PVLDB*, 7(10):903–906, 2014.
- [20] E. Wu, F. Psallidas, Z. Miao, H. Zhang, and L. Rettig. Combining design and performance in a data visualization management system. In *CIDR*, 2017.

Research Directions in Blockchain Data Management and Analytics

Hoang Tam Vo
IBM Research – Australia

Ashish Kundu
IBM Research – Yorktown Heights,
USA

Mukesh Mohania
IBM Research – Australia

ABSTRACT

Blockchain technology has emerged as a primary enabler for verification-driven transactions between parties that do not have complete trust among themselves. Bitcoin uses this technology to provide a provenance-driven verifiable ledger that is based on consensus. Nevertheless, the use of blockchain as a transaction service in non-cryptocurrency applications, for example, business networks, is at a very nascent stage. While the blockchain supports transactional provenance, the data management community and other scientific and industrial communities are assessing how blockchain can be used to enable certain key capabilities for business applications.

We have reviewed a number of proof of concepts and early adoptions of blockchain solutions that we have been involved spanning diverse use cases to draw common data life cycle, persistence as well as analytics patterns used in real-world applications with the ultimate aim to identify new frontier of exciting research in blockchain data management and analytics. In this paper, we discuss several open topics that researchers could increase focus on: (1) leverage existing capabilities of mature data and information systems, (2) enhance data security and privacy assurances, (3) enable analytics services on blockchain as well as across off-chain data, and (4) make blockchain-based systems active-oriented and intelligent.

1 INTRODUCTION

Blockchain (a.k.a. distributed ledger [31]) is an emerging platform that is designed to support transactions services within a multi-party business network, with the goal of enabling significant cost and risk reductions for all parties through the creation of innovative new business models. Data maintained within the distributed ledger can only be accessed through the execution of a smart contract [29] (i.e., a stored procedure call on the distributed ledger) that describes rules that govern a transaction. In addition, the design of blockchain technology ensures that no one business entity can modify, delete, or even append any record to the ledger without the consensus from other business entities in the network, making the system useful for ensuring the immutability of data and legal documents.

Given the aforementioned important features, blockchain technology has taken the world by storm in the recent years for its promise to transform every industry. For instance, it has started to be used in a wider range of applications, e.g., Internet of Things (IoT) [11]. A blockchain enables IoT devices to send data for inclusion in a shared transaction repository with tamper-resistant records, and enables business parties to access and supply IoT data without the need for central control and management. Blockchain for IoT can optimize supply chains by

tracking objects as they traverse the export/import supply chain while enforcing shipping and expediting incremental payments.

Similarly, blockchain technology also has the potential to disrupt insurance industry. It has been used in a car micro-insurance application to enable the concept of pay-as-you-go insurance [20]. This application allows drivers who rarely use cars to only pay insurance premium for particular trips rather than the hefty yearly premium. By transparently storing on blockchain all the data pertaining to the actual trip and premium payment, every party in the insurance contract including the driver, the insurance company, and the financial institution is confident that the data are tamper-proof and traceable. This guarantees that any insurance claim request regarding to a trip can be processed quickly and indisputably, hence offering a better customer experience. Further, as the micro-insurance application also requires accessing multiple risk analytic databases such as past driving behaviour statistics and past vehicle runtime statistics for computing premiums, a system architecture that allows for maintaining and analyzing both on-chain and off-chain data was also proposed.

In fact, Gartner's 2016 research report¹ identified blockchain as one of the key platform-enabling technologies to track. Nonetheless, while there is currently no standard in the blockchain space, there is a growing consensus that blockchain is entering its peak of inflated expectations. The report anticipated that it would take 5 to 10 years for blockchain technology to get mainstream adoption. Further, most of nowadays blockchain efforts, especially when applied to business environments, are still in a nascent state. Research perspectives and challenges related to blockchain have been presented in [12], but they are mainly for cryptocurrencies and public blockchain environments. The time is ripe for database community to get more deeply involved in solving open problems pertaining to data management and analytics in a permissioned blockchain network for business applications [28].

As the building blocks of blockchain include some combination of database, transaction, encryption, consensus and other distributed system technologies, it is natural to investigate if it is possible to utilize existing capabilities of mature data and information systems through robust integration into blockchain systems. There exist open research issues such as multi-storage and index support, novel transaction concurrency model, scalable transaction throughput, master and reference data management, smart contract management, data security and privacy assurances, as well as information leakage prevention.

Furthermore, even though the blockchain database is useful for transparent persistence of streaming business data, there is no one-size-fits-all database solution for an application [30]. While blockchain is originally designed to maintain transaction data, there is a growing interest in providing analytics capabilities in blockchain-based data systems. Particularly, in this paper we shall elaborate on specific research problems such as built-in analytics for blockchain, and data integration and analytics across on-chain and off-chain data.

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

¹<http://www.gartner.com/newsroom/id/3412017>

Last but not least, with recent advances in areas such as information retrieval, machine learning, and AI, there is a tremendous opportunity to bring cognitive capabilities, e.g., understanding, learning, and contextual awareness into blockchain-based data systems so as to make them active-oriented and intelligent. We shall discuss open research issues encountered while developing intelligent blockchain-based data systems. Our list is by no means comprehensive and other research opportunities exist as well.

2 BACKGROUND

Blockchain. Blockchain technology [31] provides a framework for building a distributed ledger that can provide consensus, provenance, immutability and finality of transaction data. The use of blockchain was first popularised by Bitcoin [1], which is a cryptocurrency. In a blockchain, a group of ledger entries, i.e., a list of transactions, are periodically accumulated into a block which contains a cryptographic hash of the prior block linking the blocks together. This way of chaining the blocks allows the global order of the ledger entries to be established and to verify that the content of a particular block have not been modified. Every node in this distributed system maintains its own copy of the blockchain and participates in an appropriate consensus mechanism to keep the replicated data in sync across nodes. For example, Bitcoin uses a consensus protocol called “Proof-of-Work” [31], whereas Hyperledger Fabric [8] develops a variant of Byzantine fault-tolerant (BFT) state machines [32].

Permissioned/private blockchain. In Bitcoin, a public implementation of blockchain, entities that participate in the transfer of assets are anonymous and any entity can participate. In contrast, many business networks may have a need for a distributed ledger that is only accessible to a closed community of known entities. Permissioned blockchain technologies such as Hyperledger Fabric[8] and R3 Corda[5] have been developed to support these requirements, i.e., entities participating the network are identified so that their permissions can be determined and the activities of an entity are only visible to those participants of the business network that have a need to know.

Smart contract. Some distributed ledger technologies support an additional capability called a smart contract [29], which is similar to the concept of stored procedure in classical relational databases to some extent. Smart contracts allow the shared business processes within a business network to be standardised, automated and enforced via computer programs to increase the integrity of the ledger.

3 BLOCKCHAIN DATA MANAGEMENT

3.1 Leverage capabilities of mature data and information systems

Multi-storage and index support. Most blockchain platforms such as Ethereum [7] adopt key-value data model, while a few of them like R3 Corda [5] use relational data model. This makes any single blockchain platform not suited for different types of data used in a wide range of business applications. For example, geo-location data recorded from vehicles in a car micro-insurance application [20] as discussed in Section 1 may not be efficiently queried using a key-value store. Furthermore, even though blockchain platforms such as Hyperledger Fabric [8] opt for pluggable storage model, developer users have to decide at development time which storage to use, e.g., either LevelDB [9] (key-value store) or CouchDB [6] (document store). Therefore, novel techniques are needed for supporting multiple types of

data stores such as key-value, document, SQL and spatial data stores simultaneously in the same blockchain system.

Additionally, blockchain is originally not designed to store digital documents, which, however, are a popular type of data shared in a business network as observed in a majority of blockchain solutions that we have been involved. These digital records are usually large, and their aggregate size grows significantly over time. It is infeasible to store these data directly on the blockchain due to several constraints such as storage size, bandwidth and transaction throughput. One possible solution is to store these records in a third-party offline storage, and maintain their locations and a digital hash of the data on the blockchain for verification. Nevertheless, this approach requires integration of blockchain and offline storages. Thus, it is critical to develop blockchain systems with built-in offline storage strategies for handling big data.

We also observe from the implementation of those blockchain solutions that rich queries (e.g., conditionals, operators etc.) of data on blockchain are typically read-only and based on non-primary keys. To deal with these situations, explicit smart contracts for maintaining secondary indices have to be developed. This motivates exciting research problems related to index management in blockchain-based data systems.

Master data management. Unlike blockchains used in public cryptocurrency environments, a business blockchain network is not a single universal collaborative environment for every organization to join in this same network. Instead, each network usually includes a specific set of organizations sharing some common business interests, and more importantly, an organization may join a number of different blockchain networks due to the large scope of their business. It is likely that each network will have a different data schema and may record a different version of some common data referring to the same entity across the networks. Therefore, organizations need master data management rules, processes and techniques in order to consolidate data across multiple blockchain networks that they participate in. In addition, we also envision an interesting opportunity for future research to explore a new concept of cross-chain smart contracts that run across multiple blockchains.

Reference data management. Since the data in blockchain cross over the boundary of organizations, semantically right interpretation of data is must. Hence, one important problem is interpreting the data w.r.t. reference data and business glossary. Particular technical problems include identification of reference data entities, automatic interpretation/conversion, and managing the reference data as they are provided by external sources. Another technical question is whether this logic of references should be handled in the smart contract or at application level. Further, query processing on blockchain must take such context, i.e., references, into account to carry out meaningful processing.

Scalable transaction throughput. There continues to be the quest for scalable transaction throughput in blockchain. The blockchain in Bitcoin [1] uses “Proof of Work” (PoW) consensus method that is computationally expensive (by design) for having to solve a cryptographic puzzle in the process [12, 31]. Instead, the permissioned blockchains where participants are identified use consensus methods based on variants of Byzantine fault-tolerant (BFT) state machines [32], which have been chosen to provide higher transaction throughput and lower consensus latency as shown in the recent benchmark [16]. Nevertheless, even with that performance improvement, the benchmark paper concludes that current blockchain technologies are not suited for large-scale data processing workloads. Consequently, there still exists the

need of novel methods, e.g., implicit consensus [22] and sharding data [19] that provide high levels of transaction throughput for blockchain. A different direction to achieve scalable throughput, e.g., BigchainDB [4], is developing blockchain-like trusted transactions on top of existing modern distributed database systems.

ACID properties. Presently, no blockchain platform fully guarantees ACID properties [24] because blockchain is not designed to support databases, nor it is always tractable to support these properties on distributed ledgers. Nevertheless, as we apply transactional semantics to blockchain and use it for data management, we need to assure the important ACID properties. It is interesting to see that even though blockchain platforms are designed to maintain transaction data, they adopt a simple concurrency control, or not at all, to deal with concurrent transactions accessing the same data item. Transactions in blockchains are validated based on a first-come first-served basis. That is, the first transaction to get endorsed and committed by all nodes in the blockchain network wins and invalidates other conflicting transactions that are concurrently modifying some common data items. In this case, all other client applications executing those conflicting transactions have wasted time waiting for the notification from the blockchain about the rejected status of their submitted transactions. Hence, an important research issue here is to develop novel concurrency control models for blockchains so that they are applicable to a wider range of applications.

3.2 Enhance information protection

Blockchain-enabled applications involving security- and privacy-sensitive data, e.g., financial [17] and healthcare [33], requires confidentiality, security and privacy assurances at different levels to be supported by the system, which are mandated by regulatory compliance requirements such as HIPAA [2] and GDPR [3].

Confidentiality. Access control mechanism is mainly used to protect access data on permissioned blockchains [18]. However, access control is insufficient to provide protection from data exposure. Data that is stored on distributed ledger of the blockchain networks need to be encrypted. Thus, it is essential to determine the “computational hop” in which data shall remain encrypted, primarily because if we assume that data shall remain encrypted across its life cycle, processing of such data using smart contracts shall be difficult (unless we use fully homomorphic encryption or some form of malleable encryption schemes). Querying data on blockchain has to be enabled even when the data is encrypted. Models such as “search on encrypted databases” if implemented shall have severe impact on the performance of the blockchain system. Therefore, in the presence of encrypted data available for querying, the query execution system on blockchain has to be properly designed and implemented.

Privacy. Bitcoin [1] and Ethereum [7] claim to support some form of privacy for transactional information, which, however, has been shown not to be entirely privacy-preserving [23]. Data on blockchain may need to be shared across others for analytics. There are use cases where research needs to be carried out on implementing “right-to-forget” on blockchain. GDPR and EU regulation on data privacy have recently asked Google and other internet companies to support “right-to-forget” – a user may ask the service providers to remove their data from the search results or from the system altogether. When a blockchain stores healthcare, finance and such other sensitive data, a user may request the blockchain provider to delete some or all the records pertaining to the user, which is hard to support today due to

the immutability of blockchain. This necessitates cryptographic techniques to rewrite history in blockchains [10].

Information leakage prevention. For blockchains that maintain unstructured data such as business documents, a more comprehensive data redaction mechanism [15] would be needed to protect business-critical information contained in these documents from unintentional disclosure. Typically, these documents can be accessed fully or they are protected completely. However, there is real need that documents can be released partially by redacting certain data entities that should be prohibited from the users. Thus, it is important to support fine-grained access control on these documents (i.e., access control at data entity level rather than at document level). This fine-grained access control can be achieved by detecting and removing the business-critical information from the document shared on the blockchain based on the role of the user who requests to access the document.

3.3 Manage smart contracts

Smart contract governance. As smart contracts capture the shared business processes between parties in a blockchain network, the governance of these smart contracts at every step of their life cycle including business analysis, design, development, testing, deployment, and monitoring would in general require the coordination and approval of these multiple parties. Nevertheless, this governance process is not standardised and automated as per current practice. Instead, it is common that only one or a subset of parties are delegated and trusted to manage the entire life cycle of a smart contract. This points to the need of tools that allow automated and collaborative governance of smart contracts.

Smart contract template. As current practice, smart contracts are manually programmed by developers after studying requirements described in legal documents agreed by multiple parties. However, this process is time consuming and error-prone as there is currently no standard regarding the legal enforceability of code-based contracts. It is, therefore, important to make the development of smart contracts as much automated as possible. An open research issue in this direction is to define a standardised semantic framework for smart contracts that considers both operational and non-operational aspects based on existing legal documents [13]. Another related research challenge is to propose standardised templates for automated generation of legally-enforceable smart contracts from legal documents. This requires understanding natural language used in legal documents and identifying operational parameters that can be used as the connection between legal agreements and smart contracts.

Trusted smart contracts. Smart contracts vulnerable to attacks can expose blockchain data or contain backdoors that may be used to exfiltrate data from the blockchain. Hence, it is essential to develop security analysis technology for smart contracts and reasoning about their semantic trustworthiness, i.e., trust between parties defined based on the semantics of smart contracts rather than the digital signatures of the code for smart contracts. Several open questions need to be addressed, e.g., how to detect bugs in smart contract codes [26], how to deal with smart contract codes behaving erratically and how to update with correct codes without impacting the network [27].

4 BLOCKCHAIN DATA ANALYTICS

4.1 Built-in analytics for blockchain

As the original blockchain is purely a transaction repository, an execution engine will be required for analytics running directly

on blockchain data. A possible solution to this problem is to make blockchain data readily accessible by data parallel processing systems such as MapReduce [14] or Spark [34]. In particular, an input reader could be implemented so that MapReduce and Spark programs are able to scan through blockchain data efficiently. Further, MapReduce or Spark execution nodes can be physically co-located with blockchain data nodes to reduce the need of data transfer, and hence improving analytics performance. Apart from the above batch analytics, there are also use cases such as IoT applications in the supply chain domain as discussed in Section 1 in which lightweight or edge analytics capability (i.e., analyzing data as ingested into blockchain) would be critical to the system.

4.2 Integration and analytics across on-chain and off-chain data

It is worth noting that the need of data integration across multiple blockchains that an organization participates in, as discussed in Section 3.1, is just one dimension of the problem. Another dimension of data integration problem comes from common data entities referred by both the blockchains and the organization's legacy systems of record. In particular, whereas blockchains function independently of legacy systems in most cases, at some point in the application development process organizations will need to integrate blockchain data with their existing systems of record for deriving complete business insights. Since multiple parties are joining a blockchain network, cases of overlapping or inconsistent data between the blockchain and their legacy systems will likely arise. As a consequence, there is much scope for development of new techniques in entity resolution for big data spanning across blockchain and off-chain data.

In addition, as the analytics now spans across on-chain and off-chain data systems, query processing over federated data and optimisation techniques would be the key to the performance of query federation. For example, would the strategy that exports all data on blockchain into an off-chain database where all the analytics are executed be optimal? In contrast, are there better approaches that only materialize part of relevant blockchain data in the off-chain database and how it could be done dynamically given the changing workload? More importantly, it is also challenging to ensure the immutability of the data that has been exported from blockchain into external data stores. These are very interesting research issues and they require more exploration on query federation, translation, and optimization, as well as data security in the context of analytics over both on-chain and off-chain data.

5 INTELLIGENT BLOCKCHAIN SYSTEMS

As discussed in Section 1, blockchain technology has started to be used in a wider range of applications, e.g., Internet of Things (IoT) [11]. Nevertheless, the volume of data generated in this era of the Internet of Things is growing significantly, which puts blockchain systems to their limits of transaction throughput and storage capacity. Consequently, when a new piece of data arrives, it is important for the blockchain system to be able to understand the input data, reason about its relevance to the business so as to determine whether dropping the data or accepting and storing it in the blockchain. Recently, a technique to reduce data acquisition cost by only accepting data that is useful for answering queries has been proposed [25]. However, none of prior systems is able to self learn the relevance of incoming data to the business. This necessitates an active-oriented and intelligent

blockchain system for making sense and intelligently classifying incoming data, which greatly helps reduce redundant data storage and computation at later stages. In fact, intelligence can be embedded at every step in the pipeline of data processing inside a blockchain-based systems, similar to the concept of intellectual data warehousing [21].

6 CONCLUSIONS

We have highlighted research topics that characterize the common issues of on-going data management and analytics problems encountered in the development of real-world blockchain applications. We hope that this study could provide a basis for further research to identify likely solutions to these open problems.

REFERENCES

- [1] 2008. Bitcoin: A Peer-to-Peer Electronic Cash System. <https://bitcoin.org/bitcoin.pdf>. (2008).
- [2] 2013. Health Insurance Portability and Accountability Act of 1996 (HIPAA). <https://www.hhs.gov/hipaa/for-professionals/index.html>. (2013).
- [3] 2016. Directive 95/46/EC (General Data Protection Regulation). <http://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX%3A32016R0679>. (2016).
- [4] 2017. BigchainDB. <https://www.bigchaindb.com/>. (2017).
- [5] 2017. Corda. <https://github.com/corda/corda>. (2017).
- [6] 2017. CouchDB. <http://couchdb.apache.org/>. (2017).
- [7] 2017. Ethereum. <https://www.ethereum.org/>. (2017).
- [8] 2017. Hyperledger. <https://www.hyperledger.org/>. (2017).
- [9] 2017. LevelDB. <https://github.com/google/leveldb>. (2017).
- [10] Giuseppe Ateniese, Bernardo Magri, Daniele Venturi, and Ewerton Andrade. 2017. Redactable Blockchain - or - Rewriting History in Bitcoin and Friends. In *Proc. of IEEE European Symposium on Security and Privacy*. 111–126.
- [11] Marcella Atzori. 2017. Blockchain-Based Architectures for the Internet of Things: A Survey. <https://ssrn.com/abstract=2846810>. (2017).
- [12] Joseph Bonneau et al. 2015. SoK: Research Perspectives and Challenges for Bitcoin and Cryptocurrencies. In *Proc. of IEEE SSP*. 104–121.
- [13] Christopher Clack et al. 2016. Smart Contract Templates: foundations, design landscape and research directions. *CoRR abs/1608.00771* (2016).
- [14] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. of OSDI*. 10–10.
- [15] Prasad Deshpande et al. 2015. The Mask of ZoRRo: preventing information leakage from documents. *KAIS* 45, 3 (2015), 705–730.
- [16] Tien Tuan Anh Dinh et al. 2017. BLOCKBENCH: A Framework for Analyzing Private Blockchains. In *Proc. of SIGMOD*. 1085–1100.
- [17] A. Kosba et al. 2016. Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts. In *Proc. of IEEE SSP*. 839–858.
- [18] Danny Yang et al. 2017. Survey of Confidentiality and Privacy Preserving Technologies. (2017). R3 Research.
- [19] Eleftherios Kokoris-Kogias et al. 2017. OmniLedger: A Secure, Scale-Out, Decentralized Ledger. *Cryptology ePrint Archive*, Report 2017/406. (2017).
- [20] Hoang Tam Vo et al. 2017. Blockchain-based Data Management and Analytics for Micro-insurance Applications. In *Proc. of CIKM*. 2539–2542.
- [21] Mukesh Mohania et al. 2018. Active, Real-Time, and Intellectual Data Warehousing. In *Encyclopedia of Database Systems*. 1–10. To appear at https://doi.org/10.1007/978-1-4899-7993-3_8-3.
- [22] Zhijie Ren et al. 2017. Implicit Consensus: Blockchain with Unbounded Throughput. *CoRR abs/1705.11046* (2017).
- [23] Michael Fleider et al. 2014. Bitcoin Transaction Graph Analysis. (2014).
- [24] Theo Haerder and Andreas Reuter. 1983. Principles of Transaction-oriented Database Recovery. *ACM Comput. Surv.* 15, 4 (1983), 287–317.
- [25] Zheng Li and Tingjian Ge. 2016. Stochastic Data Acquisition for Answering Queries as Time Goes by. *PVLDB* 10, 3 (2016), 277–288.
- [26] Loi Luu et al. 2016. Making Smart Contracts Smarter. In *Proc. of ACM CSC*. 254–269.
- [27] Bill Marino and Ari Juels. 2016. *Setting Standards for Altering and Undoing Smart Contracts*. 151–166.
- [28] C. Mohan. 2017. Blockchains and Databases. *PVLDB* 10, 12 (2017), 2000–2001.
- [29] Pablo Seijas et al. 2016. Scripting smart contracts for distributed ledger technology. *IACR Cryptology ePrint Archive* 2016 (2016), 1156.
- [30] Michael Stonebraker and Ugur Cetintemel. 2005. One Size Fits All: An Idea Whose Time Has Come and Gone. In *Proc. of ICDE*. 2–11.
- [31] Florian Tschorsch and Bjorn Scheuermann. 2016. Bitcoin and Beyond: A Technical Survey on Decentralized Digital Currencies. *IEEE Communications Surveys and Tutorials* 18, 3 (2016), 2084–2123.
- [32] Marko Vukolić. 2016. *The Quest for Scalable Blockchain Fabric: Proof-of-Work vs. BFT Replication*. 112–125.
- [33] Xiao Yue et al. 2016. Healthcare Data Gateways: Found Healthcare Intelligence on Blockchain with Novel Privacy Risk Control. *Journal of Medical Systems* 40, 10 (2016), 1–8.
- [34] Matei Zaharia et al. 2012. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proc. of NSDI*. 2–2.

Scalable Active Temporal Constrained Clustering

Son T. Mai, Sihem Amer-Yahia, Ahlame Douzal Chouakria
 CNRS, Univ. Grenoble Alpes, France
 {mtson,sihem.amer-yahia,ahlame.douzal}@univ-grenoble-alpes.fr

ABSTRACT

We introduce a novel interactive framework to handle both instance-level and temporal smoothness constraints for clustering large temporal data. It consists of a constrained clustering algorithm which optimizes the clustering quality, constraint violation and the historical cost between consecutive data snapshots. At the center of our framework is a simple yet effective active learning technique for iteratively selecting the most informative pairs of objects to query users about, and updating the clustering with new constraints. Those constraints are then propagated inside each snapshot and between snapshots via constraint inheritance and propagation to further enhance the results. Experiments show better or comparable clustering results than existing techniques as well as high scalability for large datasets.

1 INTRODUCTION

In semi-supervised clustering, domain knowledge is typically encoded in the form of instance-level *must-link* and *cannot-link* constraints [8]. Such constraints specify that two objects must be placed in the same clusters or not. Constraints have been successfully applied to improve clustering quality in real-world applications, e.g., identifying people from surveillance cameras [8] and aiding robot navigation [7]. However, current research on constrained clustering still suffers from several issues.

Most existing approaches assume that we have a set of constraints beforehand, and an algorithm will use this set to produce clusters [2, 7]. Davidson et al. show that the clustering quality varies significantly using different equi-size sets of constraints [5]. Moreover, annotating constraints requires human intervention, an expensive and time consuming task that should be minimized as much as possible given the same expected clustering quality. Therefore, how to choose a *good* and *compact* set of constraints rather than randomly selecting them from the data has been the focus of many research efforts, e.g., [1, 11, 14].

Many approaches employ different *active learning* schemes to select the most meaningful pairs of objects and then query experts for constraint annotation [1, 11]. By allowing the algorithms to choose constraints themselves, we can avoid insignificant ones, and expect to have high quality and compact constraint sets compared to randomized constraint selection. These constraints are then used as input for constrained clustering algorithms to operate. However, if users are not satisfied with the results, they are asked to provide another constraint set and start the clustering again, which is obviously time consuming and expensive.

Other algorithms follow a *feedback* schema which does not require a full set of constraints in the beginning [4]. They iteratively produce clusters with their available constraints, show results to users, and get feedback in the form of new constraints. By iteratively refining clusters according to user feedback, the acquired results fit users' expectations better [4]. Constraints

are also easier to select with an underlying cluster structure as a guideline, thus reducing the overall number of constraints and human annotation effort for the same quality level. However, exploring the whole data space for finding meaningful constraints is also a non-trivial task for users.

To reduce human effort, several methods incorporate *active learning* into the feedback process, e.g., [10, 11, 14]. At each iteration, the algorithm automatically chooses pairs of objects and queries users for their feedback in terms of *must-link* and *cannot-link* constraints instead of leaving the whole clustering results for users to examine. Though these active feedback techniques are proven to be very useful in real-world tasks such as document clustering [10], they suffer from very high runtime since they have to repeatedly perform clustering as well as exploring all $O(n^2)$ pairs of objects to generate queries to users.

In this paper, we develop an efficient framework to cope with the above problems following the iterative active learning approach as in [10, 14]. However, instead of examining all pairs of objects, our technique, called *Border*, selects a small set of objects around cluster borders and queries users about the most uncertain pairs of objects. We also introduce a constraint inheritance approach based on the notion of μ -nearest neighbors for inferring additional constraints, thus further boosting performance. Finally, we revisit our approach in the context of evolutionary clustering. Evolutionary clustering aims to produce high quality clusters while ensuring that the clustering does not change dramatically between consecutive timestamps. We propose to formulate a temporal smoothness constraint and add a time-fading factor to our constraint propagation.

This paper's contributions are: (i) a new algorithm CVQE+ that extends CVQE [7] with weighted *must-link* and *cannot-link* constraints, (ii) a new algorithm, *Border*, that relies on active clustering and constraint inheritance to choose a small number of objects to solicit user feedback for, (iii) an evolutionary clustering framework which incorporates instance-level and temporal smoothness constraints, and (iv) experiments with 6 datasets that show the superiority of our algorithms over state-of-the-art ones.

2 PROBLEM FORMULATION

Let $D = \{(d, t)\}$ be a set of $|D|$ vectors $d \in \mathbb{R}^p$ observed at time t . Let $S = \{(S_s, D_s, ts_s, te_s)\}$ be a set of preselected $|S|$ data snapshots. Each S_s starts at time ts_s , ends at time te_s and contains a set of objects $D_s = \{(d, t) \in D \mid ts_s \leq t < te_s\}$. Two snapshots S_s and S_{s+1} may overlap but must satisfy the time order, i.e., $ts_s \leq ts_{s+1}$ and $te_s \leq te_{s+1}$. For each snapshot S_s , let $ML_s = \{(x, y, w_{xy})\}$ be a set of *must-link* constraints related to $x, y \in D_s$ with a degree of belief of $w_{xy} \in [0, 1]$. Similarly, let $CL_s = \{(x, y, w_{xy})\}$ be a set of *cannot-link* constraints of S_s . Initially, ML_s and CL_s can be empty.

In this paper, we focus on the problem of grouping objects in all snapshots into clusters. Our goals are (1) reduce the number of constraints thus reducing the constraint annotation costs (2) make the algorithm scale well with large datasets and (3) smooth the gap between clustering results of two consecutive snapshots, i.e., ensure temporal smoothness.

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

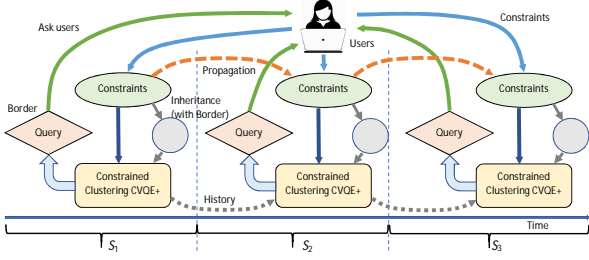


Figure 1: Our active temporal clustering framework

3 OUR PROPOSED FRAMEWORK

Figure 1 illustrates our framework which relies on two algorithms, Border and CVQE+. Our framework starts with a small (or empty) set of constraints in each snapshot. Then, it iteratively produces clustering results and receives refined constraints from users in the next iterations. This process is akin to feedback-driven algorithms for enhancing clustering quality and reducing human annotation effort [4]. However, instead of passively waiting for user feedback as in [4], our algorithm, Border, *actively* examines the current cluster structure, selects β pairs of objects whose labels are the least certain, and asks users for their feedback in terms of instance-level constraints. Examining all possible pairs of objects to select queries is time consuming due the quadratic number of candidates. To ensure scalability, Border limits its selection to a small set of most promising objects. When there are new constraints, instead of reclustering from scratch as in [10, 14], our algorithm, CVQE+, incrementally updates the cluster structures. We also aim to ensure a smooth transition between consecutive clusterings [3]. We additionally introduce two novel concepts: (1) the *constraint inheritance* scheme for automatically inferring more constraints inside each snapshot and (2) the *constraint propagation* scheme for propagating constraints between different snapshots. These schemes help significantly reduce the number of constraints for acquiring a desired level of clustering quality. To the best of our knowledge, Border is the first framework that combines active learning, instance-level and temporal smoothness constraints.

The new algorithm CVQE+. For each snapshot S_s , we use constrained kMeans for grouping objects. Any existing techniques such as MPCK-Means [2], CVQE [7] or LCVQE [13] can be used. Here we introduce CVQE+, an extension of CVQE [7] to cope with weighted constraints, to do the task. Let $C = \{C_i\}$ be a set of clusters. The cost of C_i is defined as its vector quantization cost VQE_i and the constraint violation costs ML_i and CL_i as follows. Note that, our ML_i cost is symmetric compared to [7].

$$\begin{aligned} Cost_{C_i} &= Cost_{VQE_i} + Cost_{ML_i} + Cost_{CL_i} \quad (1) \\ Cost_{VQE_i} &= \sum_{x \in C_i} (c_i - x)^2, \\ Cost_{ML_i} &= \sum_{(a,b) \in ML_i \wedge vl(a,b)} w_{ij} (c_i - c_{\pi(a,b,i)})^2 \\ Cost_{CL_i} &= \sum_{(a,b) \in CL_i \wedge vl(a,b)} w_{ij} (c_i - c_{\varphi(i)})^2 \end{aligned}$$

where, $vl(a,b)$ is true for (a,b) that violates *must-link* or *cannot link* constraints, c_i is the center of cluster C_i , $\pi(a,b,i)$ returns the center of clusters of a or b (not including cluster C_i), and $\varphi(i)$ returns the nearest cluster center of C_i . Note that, $Cost_{ML_i}$ is symmetric compared to [7]. Taking the derivative of $Cost_{C_i}$, the

new center of C_i is updated as:

$$c_i = \frac{\sum_{x \in C_i} x + \sum_{(a,b) \in ML_i \wedge vl(a,b)} w_{ij} C_{\pi(a,b,i)} + \sum_{(a,b) \in CL_i \wedge vl(a,b)} w_{ij} C_{\varphi(i)}}{|C_i| + \sum_{(a,b) \in ML_i \wedge vl(a,b)} w_{ij} + \sum_{(a,b) \in CL_i \wedge vl(a,b)} w_{ij}} \quad (2)$$

For each constraint (a,b) , CVQE+ assigns objects to clusters by examining all k^2 cluster combinations for a and b like CVQE. The major difference is that when we calculate the violation cost, we consider all constraints starting and ending at a and b instead of only the constraint (a,b) as in CVQE [6] or LCVQE [6], which is very sensitive to the cost change when some constraints share the same objects (changing these objects affects all their constraints). Thus, this scheme is expected to improve the clustering quality of CVQE+ compared to CVQE and LCVQE.

Active learning with Border. To avoid examining all pairs of objects, Border chooses a subset of $m = \min(O(\sqrt{n}), M)$ objects located at the boundary of the clusters as the main targets since they are the most uncertain ones, where M is a predefined constant. For each object a in cluster C_i , the border score of a is defined as $bor(a) = \frac{(a-c_i)^2}{(a-c_{\varphi(i)})^2(1+ml(a))(1+cl(a))}$, where $ml(a)$ and $cl(a)$ are the sums of weights of must and cannot-link constraints of a . Here, we favor objects that have fewer constraints for increasing constraint diversity. This also fits well with our constraint inheritance scheme. For each cluster C_i , we select $m|C_i|/n$ top objects based on their border score distribution in C_i . This can be done by building a histogram with $O(\sqrt{|C_i|})$ bins (a well-known rule of thumb for the optimal histogram bin). Then, objects are taken sequentially from the outermost bins.

For each selected object a , we estimate the uncertainty of a as $sco(a) = ent(\mu nn(a)) + \frac{vl(ml(a))+vl(cl(a))}{ml(a)+cl(a)+1}$, where $ent(\mu nn(a))$ is the entropy of class labels of μ nearest neighbors of a and $vl(ml(a))$ and $vl(cl(a))$ are the sums of violated must-link and cannot-link constraints of a . A high $sco(a)$ means that a is in high uncertain areas with different mixed class labels and a high number of constraint violations.

We divide $m^2 = O(n)$ pairs of selected objects into two sets: the set of inside cluster pairs X and between cluster pairs Y , i.e., for all $(x,y) \in X : label(x) = label(y)$ and for all $(x,y) \in Y : label(x) \neq label(y)$. For a pair $(x,y) \in X$, it is sorted by $val(x,y) = \frac{(x-y)^2(1+sco(x))(1+sco(y))}{(1+ml(x)+cl(x))(1+ml(y)+cl(y))}$. For $(x,y) \in Y$, $val(x,y) = \frac{(x-y)^2(1+ml(x)+cl(x))(1+ml(y)+cl(y))}{(1+sco(x))(1+sco(y))}$. The larger val is, the more likely x and y belong to different clusters and vice versa. We choose top $\beta/2$ non-overlapped largest val pairs of X and top $\beta/2$ non-overlapped smallest pairs of Y in order to maximize the changes in clustering results (inside and between clusters).

We show β pairs to users to ask for the constraint type and add their feedback to the constraints set and update clusters until the total number of queries exceeds a predefined budget δ .

Constraint inheritance in Border. For further reducing the number of queries to users, the general idea is to infer new constraints automatically based on annotated ones. Our inheritance scheme is based on the concept of μ nearest neighbors below.

Let h be the distance between an object p and its μ nearest neighbors. The influence of p on its neighbor x is formulated by a triangular kernel function $\phi_h(p,x)$ centered at p as in Figure 2. Given a constraint (p,q,w_{pq}) , for all $a \in \mu nn(p)$ and $b \in \mu nn(q)$, we add (a,b,w_{ab}) to the constraints set, where w_{ab} is defined as:

$$w_{ab} = w_{pq} \phi_h(p,a) \phi_h(q,b) \quad (3)$$

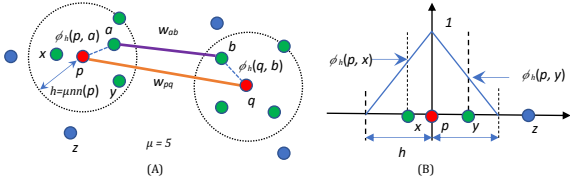


Figure 2: (A) Constraint inheritance from (p, q) to (a, b) . (B) The effect of the object b on its neighbors

The general intuition is that the label of an object a tends to be consistent with its closest neighbors (which is commonly used in classification). This scheme is expected to increase the clustering quality, especially when combined with the active learning approach described above.

Updating clusters. For incrementally updating clusters, we only need to take the old cluster centers and update them following Equation 1 with the updated constraints set.

Temporal smoothness. The general idea of temporal smoothness [3] is that clusters not only have high quality in each snapshot but also do not change much between sequential time frames. We re-define the cost of cluster C_i of snapshot S_s in Equation 1 as follows:

$$TCost_{VQE_i} = (1 - \alpha)Cost_{VQE_i} + \alpha Hist(C_i, S_{s-1}) \quad (4)$$

where $Hist(C_i, S_{s-1})$ is the historical cost of cluster C_i between two snapshots S_s and S_{s-1} and α is a regulation factor to balance the current clustering quality and the historical cost. We define the historical cost as follows:

$$Hist(C_i, S_{s-1}) = (c_i - \psi(C_i, S_{s-1}))^2 \quad (5)$$

where $\psi(C_i, S_{s-1})$ returns the closest cluster center to C_i in snapshot S_{s-1} . Taking the derivation of (4) as in (1), we have $C_i = \frac{(1-\alpha)A + \alpha\psi(C_i, S_{s-1})}{(1-\alpha)B + \alpha}$, where A and B are respectively the numerator and the denominator given in Equation 1.

Constraint propagation. Whenever we have a new constraint (x, y, w_{xy}) in snapshot S_s , we propagate it to snapshots $S_{s'}$ where $s' > s$ if $x, y \in S_{s'}$. The intuition is that if x and y are linked (either by must or cannot-link) in S_s , they are more likely to be linked in $S_{s'}$. Thus we add the constraint (x, y, w'_{xy}) to $S_{s'}$ where:

$$w'_{xy} = w_{xy} \frac{te_s - ts_{s'}}{te_{s'} - ts_s} \quad (6)$$

where $(te_s - ts_{s'}) / (te_{s'} - ts_s)$ is a time fading factor. This scheme helps to increase the clustering quality by putting more constraints into the clustering algorithm like the inheritance scheme.

Complexity analysis. Let n be the number of objects. Both CVQE+ and Border have linear time complexity to $O(n)$.

4 EXPERIMENTS

Experiments are conducted on a workstation with 4.0Ghz CPU and 32GB RAM using Java. We use 6 datasets Iris, Ecoli, Seeds, Libras, Optdigits and Wdbc acquired from the UCI archives¹. The numbers of clusters k are acquired from the ground truths. We use Normalized Mutual Information (NMI) [12] for assessing the clustering quality. All results are averaged over 10 runs.

Performance of CVQE+. Figure 3 shows comparisons among CVQE+ and existing techniques including kMeans, MPCK-Means [2], CVQE [7] and LCVQE [13]. CVQE+ consistently outperforms or acquires comparable results to CVQE and others, especially when the number of constraints is large. This can be explained

¹<http://archive.ics.uci.edu/ml/>

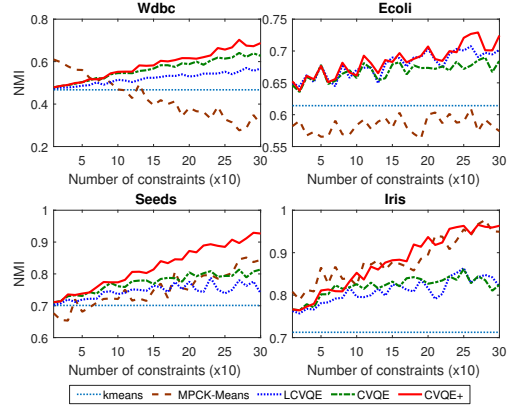


Figure 3: Performance of CVQE+ compared to others

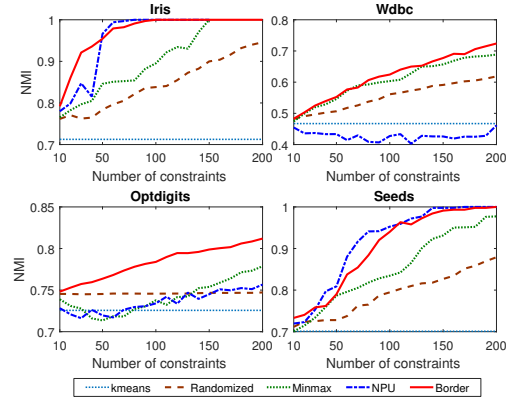


Figure 4: Comparison among different active learning techniques

by the way CVQE+ assigns objects to clusters. Compared to its predecessor algorithm CVQE or LCVQE, it deals well with constraint overlap (constraints that share the same objects), which increases with the number of constraints.

Active constraint selection. Unless otherwise stated, the budget limitation δ is set to 200, the query size $\beta = 10$ and the neighborhood size $\mu = 4$. Figure 4 shows comparisons between Border, NPU [14], Huang [14] (a modified version of [10] for working with non-document data), Min-max [11], Explorer-Consolidate [1], and a randomized method (Huang and Consolidate are removed from Figure 4 for readability). Border acquires better results than others on Libras, Wdbc and Optdigits, comparable results on Iris and Ecoli. For the Seeds dataset, it is outperformed by NPU. The difference is because Border tends to strengthen existing clusters by fortifying both the cluster borders and inter connectivity for groups of objects rather than connecting a single object to existing components like others. However, Border has some parameters to set such as the query size β . Tuning these parameters is a difficult problem that requires deeper investigation.

For runtimes, we create five synthetic datasets of sizes 2000 to 10000 consisting of 5 Gaussian clusters and measure the time for acquiring 100 constraints. Border is orders of magnitude faster than others in selecting pairs to query. For 1000 objects, it takes Border 0.1 seconds while NPU and Min-max need 439.4 and 3.0 seconds. For 10000 objects, Border, NPU and Min-max consumes 0.18, 5216.3 and 18.2 seconds, respectively. Additionally, the higher the number of objects and constraints, the higher the runtime differences. We omit the plots due to space limitations.

Cluster update. Figure 5 shows the NMI and the number of iterations of our algorithm for the Ecoli dataset. The NMI scores

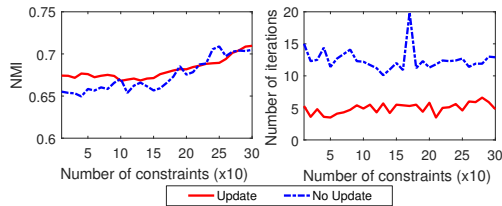


Figure 5: Update vs. fully reclustering for the Ecoli dataset

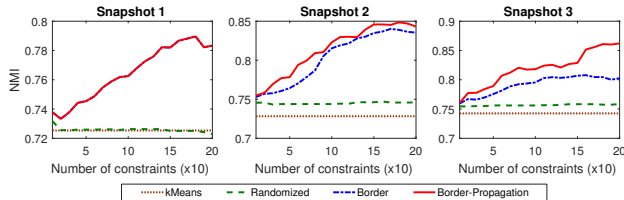


Figure 6: Temporal clustering on the dataset Optdigits

are comparable, while it takes fewer iterations for our algorithm to converge in its update mode.

Temporal clustering. Figure 6 shows the active temporal clustering results for three snapshots of the Optdigits dataset (we set $\alpha = 0.5$). As we see, our active learning scheme can help boost clustering quality inside each snapshot compared to the original kMeans or a randomized constraint selection method. With the constraint propagation scheme (Border-Propagation), the clustering results are further boosted compared to Border. Since we only consider forward propagation, the clustering result in Snapshot 3 will be more affected than Snapshot 2 and Snapshot 1. We can easily extend the algorithm for the backward propagation case.

5 RELATED WORK

Constraint clustering. There are many proposed constrained clustering algorithms such as MPC-kMeans [2], CVQE [7] and LCVQE [13]. These techniques optimize an objective function consisting of the clustering quality and the constraint violation cost like our algorithm CVQE+. CVQE+ is an extension of CVQE [7], where we extend the cost model to deal with weighted constraints, make the must-link violation cost symmetric and change the way each constraint is assigned to clusters by considering all of its related constraints. This makes cluster assignment more stable, thus enhancing the clustering quality. Interested readers are referred to [6] for a comprehensive survey on constrained clustering methods.

Active learning. Most existing techniques employ *active learning* for acquiring a desired constraints set before or during clustering. In [1], the authors introduce the Explorer-Consolidating algorithm to select constraints by exploiting the connected-components of must-link ones. Min-max [11] extends the Consolidation phase of [1] by querying most uncertain objects rather than randomly selecting them. These techniques produce constraints sets before clustering. Thus, they cannot exploit the cluster labels for further enhancing performance. Huang et al. [10] introduce a framework that iteratively generates constraints and updates clustering results until a query budget is reached. However, it is limited to a probabilistic document clustering algorithm. NPU [14] also uses connected-components of must-link constraints as a guideline for finding most uncertain objects. Constraints are then collected by querying these objects again existing connected components like the Consolidate phase of [1]. Though more effective than pre-selection ones, these techniques typically have a quadratic runtime which makes them infeasible to cope

with large datasets like Border. Moreover, Border relies on border objects around clusters to build constraints rather than must-link graphs [1, 14]. The inheritance approach is closely related to the constraint propagation in the multi-view clustering algorithm [9] for transferring constraints among different views. The major difference is that we use the μ -nearest neighbors rather than the ϵ -neighborhoods which is limited to Gaussian clusters and can lead to an excessive number of constraints.

Temporal clustering. Temporal smoothness has been introduced in the evolution framework [3] for making clustering results stable w.r.t. the time. We significantly extend this framework by incorporating instance-level constraints, active query selections and constraint propagation for further improving clustering quality while minimizing constraint annotation effort.

6 CONCLUSION

We introduce a scalable novel framework which incorporates an iterative active learning scheme, instance-level and temporal smoothness constraints for coping with large temporal data. Experiments show that our constrained clustering algorithm, CVQE+, performs better than existing techniques such as CVQE [7], LCVQE [13] and MPC-kMeans [1]. By exploring border objects and propagating constraints via nearest neighbors, our active learning algorithm, Border, results in good clustering results with much smaller constraint sets compared to other methods such as NPU [14] and Min-max [11]. Moreover, it is orders of magnitude faster making it possible to cope with large datasets. Finally, we revisit our approach in the context of evolutionary clustering adding a temporal smoothness constraint and a time-fading factor to our constraint propagation among different data snapshots. Our future work aims at providing more expressive support for user feedback. We are currently using our framework to track group evolution of our patient data with sleeping disorder symptoms.

Acknowledgments. We thank anonymous reviewers for their valuable comments. This work is supported by the IDEX CDP LIFE Project.

REFERENCES

- [1] Sugato Basu, Arindam Banerjee, and Raymond J. Mooney. 2004. Active Semi-Supervision for Pairwise Constrained Clustering. In *SDM*. 333–344.
- [2] Mikhail Bilenko, Sugato Basu, and Raymond J. Mooney. 2004. Integrating constraints and metric learning in semi-supervised clustering. In *ICML*.
- [3] Deepayan Chakrabarti, Ravi Kumar, and Andrew Tomkins. 2006. Evolutionary clustering. In *SIGKDD*. 554–560.
- [4] David Cohn, Rich Caruana, and Andrew McCallum. 2003. *Semi-supervised Clustering with User Feedback*. Technical Report.
- [5] Ian Davidson. 2012. Two approaches to understanding when constraints help clustering. In *KDD*. 1312–1320.
- [6] Ian Davidson and Sugato Basu. 2007. A Survey of Clustering with Instance Level Constraints. *TKDD* (2007).
- [7] Ian Davidson and S. S. Ravi. 2005. Clustering with Constraints: Feasibility Issues and the k-Means Algorithm. In *SDM*. 138–149.
- [8] Ian Davidson, S. S. Ravi, and Martin Ester. 2007. Efficient incremental constrained clustering. In *KDD*. 240–249.
- [9] Eric Eaton, Marie desJardins, and Sara Jacob. 2014. Multi-view constrained clustering with an incomplete mapping between views. *Knowl. Inf. Syst.* 38, 1 (2014), 231–257.
- [10] Ruizhang Huang and Wai Lam. 2007. Semi-supervised Document Clustering via Active Learning with Pairwise Constraints. In *ICDM*. 517–522.
- [11] Pavan Kumar Mallapragada, Rong Jin, and Anil K. Jain. 2008. Active query selection for semi-supervised clustering. In *ICPR*. 1–4.
- [12] Xuan Vinh Nguyen, Julien Epps, and James Bailey. 2009. Information theoretic measures for clusterings comparison: is a correction for chance necessary?. In *ICML*. 1073–1080.
- [13] Dan Pelleg and Dorit Baras. 2007. *K*-Means with Large and Noisy Constraint Sets. In *ECML*. 674–682.
- [14] Sicheng Xiong, Javad Azimi, and Xiaoli Z. Fern. 2014. Active Learning of Constraints for Semi-Supervised Clustering. *IEEE Trans. Knowl. Data Eng.* 26, 1 (2014), 43–54.

Global Range Encoding for Efficient Partition Elimination*

Jeremy Chen¹, Reza Sherkat², Mihnea Andrei³, Heiko Gerwens⁴

¹University of Waterloo, ²⁻⁴SAP SE

^{1,2}Waterloo, Canada, ³Paris, France, ⁴Walldorf, Germany

¹y522chen@edu.uwaterloo.ca, ²⁻⁴firstname.lastname@sap.com

ABSTRACT

Skipping mechanisms have been extensively studied to improve query performance over large data volumes. A powerful skipping technique for in-memory columnar databases is partition elimination. The goal is to eliminate, as much as possible, *loading* physically partitioned data into memory and *probing* column partitions against queries. This is achieved by consulting column partition summaries. The summary is often very compact compared to the column partition itself, and is kept in memory, e.g. the MINMAX zone map. These summaries have been extensively integrated into modern in-memory database systems including SAP HANA [6]. In this paper, we argue that probing by MINMAX range is not efficient when there are gaps in the values that appear in a column partition. Any predicate that needs to probe values in a gap inside a MINMAX range naturally ends up requiring a candidate check; this reduces the benefits of column partition pruning. To address this problem, we propose a mechanism to encode each partition (likewise, query) using global ranges, carefully designed to reduce false positive rates. Our approach not only provides a compact in-memory representation, but also supports efficient partition pruning using bitwise operations. Compared to MINMAX, our experiments support that our approach significantly reduces the false positive rate. It can allocate memory budget among ranges in partition groups, based on column density, estimated false positive rates from recent workload, and gaps.

1 INTRODUCTION

Partition elimination is a powerful technique to improve query performance over large volume of data [1, 4, 6, 8, 9]. To increase parallelism and achieve operational scalability, physical partitioning is often employed to divide data into independent partitions. This is done to eliminate loading partitions, and probing them against the query, when partitions to access can be inferred explicitly from the query itself. For an in-memory database, this can prevent unnecessary loads of cold partitions (better memory utilization) and can bring significant performance improvement [6]. Small materialized aggregates for partitions, e.g. the MINMAX synopsis, are small memory footprint objects that are good candidate for partition examination [5, 6]. If the predicate range of a query does not intersect with the partitions's MINMAX synopsis, then it is safe to skip the column partition without incurring false negative. Pruning by partition synopsis is effective if:

- it causes no false negative (i.e. synopsis freshness [6]), and
- it minimizes the need for redundant partition examination.

This paper considers the second synopsis requirement.

*This work was done during Jeremy's internship at SAP Labs, Waterloo, Canada.

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

2 MOTIVATION: THE LIMITATIONS OF MINMAX SYNOPSIS FOR SPARSE DATA

MINMAX synopsis has high false positive rate for sparse partitions [7]. If the synopsis of a column partition includes data points within its min-max range that do not actually exist in the column partition, pruning by MINMAX can yield false positive. This can happen when the column partition has gaps or values from a sparse distribution. For example, for a table storing the prices of an item over the years, there are usually some gaps in the price column. Although the price is expected to increase monotonically over time, it often does not increase by smallest price unit, and naturally some gaps are present. A natural improvement of the MINMAX synopsis is to store several ranges for each column partition. This way, some gaps can be excluded from the column partition representation. However, this approach has two limitations. First, it requires storing a number of value pairs for each partition. This is space-consuming as we usually have thousands of partitions. Second, processing predicates against this extended synopsis becomes very expensive; for each partition it requires comparison against ranges that represent the partition.

3 OVERVIEW OF OUR CONTRIBUTIONS

To address the two limitations stated in Sec. 2, we propose a new list-based structure called the Global Range Table (GRT). This structure helps to construct compact synopses for single column partitions, which supports efficient partition pruning using bitwise operations. Fundamentally, the pruning approach is very similar to the MINMAX synopsis [6] and to the Adaptive Range Filters [1] in that we use value ranges to determine whether to access each partition. The key properties of our ranges are:

- (1) We construct the list of value ranges that is common across all the partitions. This facilitates probing queries against synopsis using bitwise operations, and
- (2) We incorporate recent workload knowledge into our range extraction algorithm. This is motivated by the observation that frequently-accessed values can suffer more from false positives than rarely-accessed values, if the workload characteristics does not change significantly.

We use the extracted GRT to encode column partitions; each bit of the compact encoding indicates whether the corresponding column partition contains some values within the respective GRT range. The GRT ranges can be improved to have small false positive rates, based on the knowledge learned from recent workload and the importance of value ranges. We use the same encoding approach to represent each query as a bit string. This facilitates efficient query probing on partitions. Furthermore, storing only one GRT global to all partitions in memory is much cheaper, compared to storing multiple value ranges per partition. Finally, the amortized space overhead of our approach is one compressible bit string per partition. We propose algorithms to further reduce this overhead, when the memory budget to store partition synopsis is limited (Sec. 4.2.2).

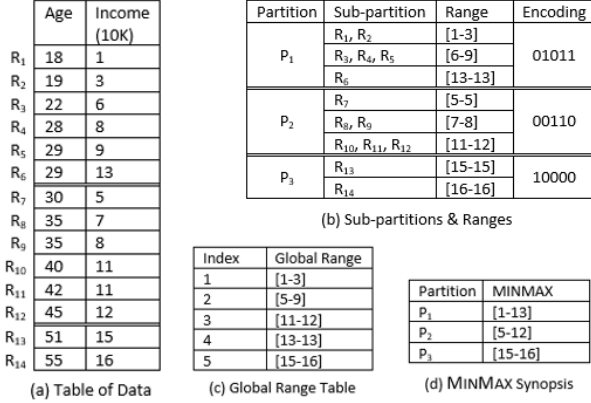


Figure 1: Encoding column partitions using Global Range Table

Table 1: List of Notations

Notation	Description
V	The universe set of column values
$C[v]$	The popularity of column value v (Sec. 4.1)
N_p	The total number of partitions
N_V^p	The total number of distinct values on the column in partition p
k	The maximum number of sub-partitions for each partition
N_R	The number of ranges in rGRT (Sec. 4.2)
m	The desired number of final GRT (i.e. the length of each encoded bit string)

4 DEEP DIVE: GLOBAL RANGE ENCODING

We demonstrate pruning by Global Range Table (GRT) using an example. Table 1 summarizes the notations used in this section.

EXAMPLE 4.1. *The table shown in Fig. 1a has two columns, Age and Income. This table is divided into three partitions based on the Age attribute (Fig. 1b). The list of ranges in Fig. 1c defines 5 mutually exclusive Income ranges. Using these Income ranges, each of the three Income column partitions in Fig. 1a can be encoded using a bit strings of length 5 in Fig. 1b. For instance, the Income column for partition P₃ is encoded as 10000. This is because this partition only contains values 15 and 16, which fall in the range [15-16] of the GRT table in Fig. 1c. As range [15-16] has index 5, only the fifth bit is set. Now consider the query $\text{Income} \leq 12$. This query intersects with three ranges from the GRT, namely [1-3], [5-9], and [11-12]. Therefore, the query can be encoded as 00111. Comparing the bit string of the query with that of the encoding column in Fig. 1c, one can verify that $00111 \wedge 10000 = 00000$, hence partition P₃ can be pruned safely. The bit string encoding preserves more gap information, compared with MINMAX, e.g. encoding P₁ as [1-13] implies that [11-12] is included. The bit string excludes this range.*

Conceptually, the global range table is a set of ranges on column partitions. The extraction of these ranges can be performed in either a single-phase process from all column partitions at once, or in a two-phase process by integrating the ranges from partitions. The two-phase approach has several advantages:

- **Memory consumption:** the single phase approach requires every column partition to be loaded into memory. The alternative way reduces the memory footprint by integrating ranges extracted from independent partitions.
- **Performance enhancement:** the two-phase approach can be implemented within the MAPREDUCE framework, with the MAP phase applied to independent partitions (Sec. 4.1) and the REDUCE step to integrate ranges (Sec. 4.2).
- **Capturing gaps:** ranges extracted from all data might not capture gaps inside partitions. Our two-step approach avoids this by regarding gaps in partitions as constraints, penalized in the goodness measure for ranges (Eq. 3).

4.1 The Sub-Partitioning Step

This step produces a set of ranges from each partition. Once the set of all ranges have been created, they will be integrated to construct GRT (Sec. 4.2). Given a set of values V , integer k , and cost function of Eq. 2, the sub-partitioning problem is to find an optimal selection of non-empty subsets S_1, \dots, S_k , s.t. $\cup_{1 \leq i \leq k} S_i = V$. As the order of values is not represented in a set, we re-order the rows in each column partition when extracting ranges and consider subsets with consecutive members. That leads to a simplified problem: given an ordered list of values V , we find the optimal k mutually exclusive sub-lists $V_1 = V[1 : i_1], \dots, V_k = V[1 + i_{k-1} : i_k]$, with $i_1 \leq j \leq k$ being indices of sorted value list.

4.1.1 *Sub-Partitioning Cost Model.* A column value is popular if it can satisfy a large fraction of queries in the recent workload¹. The more query predicates a value can satisfy, the higher its popularity ranking is. For example, suppose we have two predicates $1 \leq x \leq 5$ and $4 \leq x \leq 10$ on an integer column. Values 4 and 5 can satisfy both predicates while the other values between 1 to 10 satisfy only one predicate each. In this case, 4 and 5 are more popular than 1,2,3,6-10. Let $|Pred|$ denote the total number of predicates in the workload and let V be the set of column values. We define the popularity of a column value $v \in V$ (denoted by $C[v]$) to be the ratio of predicates from the recent workload satisfied by v . $C[v]$ is between 0 and 1, inclusive. A value is popular if it satisfies many predicates. If a popular value is in a gap within a partition, a query for this value returns nothing². Thus, we observe that the probability of false positives depends directly on the popularity of the value(s) in gaps. Let G be a gap and V_G be the set of values included in G . We define the cost of gap for G to be the sum of the popularities of the values it contains:

$$\text{Cost}(G) = \sum_{v \in V_G} C[v]. \quad (1)$$

For each sub-partition s , let V_G^s be the set of values of the gaps that are included in s . If a popular value $v \in V_G^s$ (i.e. v does not exist in sub-partition s), then those predicates that v satisfies will cause false positives on the partition to which sub-partition s belongs. The reason is that the sub-partition value range will indicate the existence of v , but it is not true (i.e. false positive). Therefore, the sub-partitioning cost of a partition is the sum of gap costs included in any created sub-partition. Let Sub_p be the set of sub-partitions in the partition p . Then,

$$\text{SubCost}(P) = \sum_{s \in \text{Sub}_p} \sum_{v \in V_G^s} C[v]. \quad (2)$$

This measure gives the likelihood that the sub-partitioning scheme will create false positives for a partition, for values that do not actually exist in the partition. Thus, creating sub-partition value ranges that include popular gaps yields a high false positive rate. Intuitively, the lower the cost is, the better the sub-partitioning scheme is. When the cost is 0, that means there is no sub-partition that produces false positives on any predicate. Thus, we would like to minimize this measure. To normalize this cost, we divide it by the maximum number of possible distinct values that the partition can take. To summarize, the normalized cost penalizes gaps that cause false positives in a set of ranges extracted from each partition. The ranges are extracted via sub-partitioning.

¹We base our approach on *predicate stability*, i.e. popular column values continue to satisfy many predicates in future queries. If predicates are not stable, our approach will remain valid but sub-optimal considering the false positive rates.

²However, a MINMAX synopsis would incur false positive on this gap.

4.1.2 The Largest Gap Greedy Algorithm. Sub-partitions with fewer gaps reduce the cost of the ranges extracted from each partition. Therefore, if we exclude popular gaps from each sub-partition, the cost would be toward optimal. The main idea of our greedy algorithm is to find top $(k-1)$ popular gaps. We use a min-heap to keep track of the top $(k-1)$ popular gaps (i.e. with largest costs). For each node of the min-heap, we keep the gap cost (i.e. Eq. 1; the gap popularity) and the index of the gap. The index of the gap is defined as the identifier of the value immediately before the gap. The min-heap property is maintained with respect to the costs stored in nodes. We go through the sorted list of distinct values of the partition and manage the top $(k-1)$ popular gaps as well as the indices of those gaps in the heap. At the end, the indices determine sub-partition boundaries. At the beginning, there are $N_V^p - 1$ candidate boundary points in partition p with N_V^p distinct values. Our algorithm selects $(k-1)$ positions to minimize the sum of the costs of sub-partitions (i.e. Eq. 2) in $O(N_V^p \log k)$ time. We omit algorithm detail and proofs for brevity.

4.2 Extracting GRT And Optimizations

The sub-partitioning step produces k value ranges per partition. Integrating at most kN_p ranges into a single list may have overlapping ranges, as well as duplicates. In this step, a value-range list global to all partitions is produced, with the primary goal to reduce false positive rates. To achieve this, we extend Eq. 2 to quantify the quality of a global range table GRT, as opposed to one partition:

$$\text{Cost}(GRT) = \sum_{r \in GRT} \sum_p \left(C[v] : v \in V_G^p \cap r \text{ s.t. } V_G^p \cap r \neq r \right). \quad (3)$$

We first integrate kN_p ranges (k ranges for N_p partitions) into one list, and call this rGRT (for raw GRT). We refine rGRT in two steps. First, we remove duplicate ranges and value range overlaps (Sec. 4.2.1). Then, for a given a memory budget, we show how to merge ranges effectively and derive a reduced list (Sec. 4.2.2).

4.2.1 Mutually Exclusive GRT. Starting from rGRT, the goal is to make every pair of ranges mutually exclusive. For this, we propose an approach based on the greedy algorithm proposed for the interval scheduling problem [3], where the range with smallest endpoint from the list is picked and inserted into the result list, if it does not intersect with any other ranges already in the result list. The range is discarded if it is in conflict with at least one range in the result list (i.e. cannot be scheduled together). In our approach, whenever the value range overlaps with some value range already in the result list, we split the to-be-inserted range into smaller ranges instead of discarding it. Each smaller range is either mutually exclusive or a duplicate (i.e. already completely covered) of the ranges in the result list. Then, we discard those duplicates and insert the others. For example, assume that we want to insert value range [2-18] to the list of ranges $R = \{[1-4], [7-10], [15-16]\}$. In our approach, we split [2-18] into six smaller ranges: [2-4], [5-6], [7-10], [11-14], [15-16], and [17-18]. This is done based on the overlap of [2-18] with the ranges in R . We process each sub-range separately. Because [2-4], [7-10], and [15-16] are already present in the result list, we discard them and insert [5-6], [11-14], and [17-18]. We iterate this split-and-insert process for every range in rGRT and obtain a mutually exclusive global range table (xGRT) in time linear to the number of ranges in rGRT. In this process, the ranges are refined and Eq. 3 reduces, but the number of ranges grows. This increases the number of bits needed to encode each partition.

4.2.2 Greedy Merge Algorithm. A synopsis is expected to have a small memory footprint. We propose a greedy algorithm to merge ranges and reduce the size of xGRT while keeping the GRT cost low. Merging t ranges R_1, \dots, R_t has two overheads: the extra range cost³ and the cost of the newly introduced gaps.

EXAMPLE 4.2. Suppose partition $P = \{1, 3, 6, 8, 9, 13\}$ and a given xGRT. Suppose we select two ranges [8-9] and [11-12] to merge, and let [11-12] be a range from another partition. The ranges [8-9] and [11-12] would merge into [8-12]. After this merge, the popularities of value 11 and 12 contribute to the GRT cost against partition P while it would not before. This is the extra cost of the range [11-12]. Before merging, the query $11 \leq x \leq 12$ would not have any match on this partition since the GRT would tell that this partition has no match. However, after merging, the GRT would suggest to load the partition since this partition has data within [8-12]. Because there is a gap between the two ranges (i.e. value 10 is missing), the cost of the gap contributes to the extra cost of merging, which comes from the popularities of 10, 11, and 12.

The greedy merge algorithm must maintain cost matrix EC , with $EC_{i,j}$ being the extra cost of merging ranges r_i, \dots, r_j in xGRT. After merging ranges r_u, \dots, r_v , all entries at row v or column v are invalidated, and every entry $EC_{i,j}$ at row u or column u must be recomputed. We continue picking the lowest extra cost and updating EC , until only m ranges remain. However, we notice that if a larger merging fully contains a smaller one, then it is guaranteed that the smaller merging has a lower (or equal) cost than the larger one since the extra-cost is non-negative. Therefore, we simplify the algorithm to use a 1-dimensional list of extra-cost, with EC having $N_R - 1$ entries. In this case, EC_i denotes the extra-cost of merging the i -th range with the next one in xGRT. At each iteration of the algorithm, we pick the lowest extra-cost, EC_u , and merge the u^{th} range with the $(u+1)^{th}$ range and update EC : we invalidate EC_{u+1} and range $(u+1)$ in xGRT. If EC_u is the last entry, then EC_u itself is invalidated. On the other hand, we at most need to update two entries of EC . If EC_u is not the first or the last entry, we re-calculate EC_{u-1} and EC_u using the updated xGRT. Note that EC_u now stores the extra-cost of merging the u^{th} and $(u+2)^{th}$ ranges as the $(u+1)^{th}$ entry has been invalidated. If EC_u is the first (the last) entry, then we only re-calculate EC_u (EC_{u-1}). We iterate until only m ranges remain. We name the result list cGRT (for compact GRT). Our algorithm takes $O((kN_p - m)N_p \log(kN_p))$ time to find cGRT.

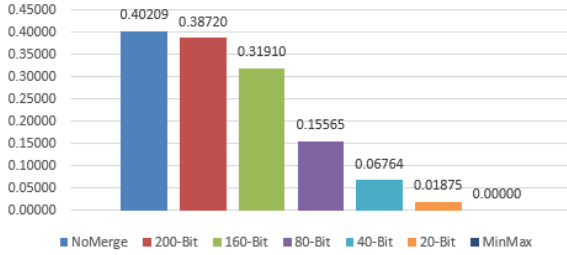
4.3 Partition Encoding and Elimination

Encoding partitions. Once the GRT is constructed with m mutually exclusive ranges, each column partition can be encoded using a bit string of length m . The bit i is 1 if and only if the partition has at least one value within the value range r_i in the GRT. This is a pre-processing step. For each partition, the encoded bit string serves as its compact synopsis, which is kept in memory.

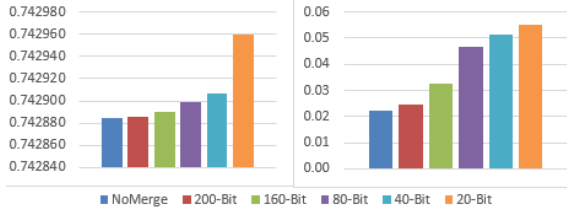
Encoding query. At query processing time, each query is encoded as a bit string. The i -th bit is set to one if at least one value in the range r_i of the GRT satisfies the query predicate.

Partition pruning. The encoded query is compared against the encoded bit string for each partition, using bitwise AND operation. If the result bit string has at least one set bit, there might be some value(s) in the corresponding partition that satisfy the query predicate. Otherwise, no record in the partition will satisfy the query predicate, and the partition can be safely pruned.

³I.e. the range cost of merging R_1, \dots, R_t but excluding R_t .



(a) GRT based encoding vs. MINMAX (ratio of gaps preserved)



(b) Normalized GRT cost, varying the number of bits (left)

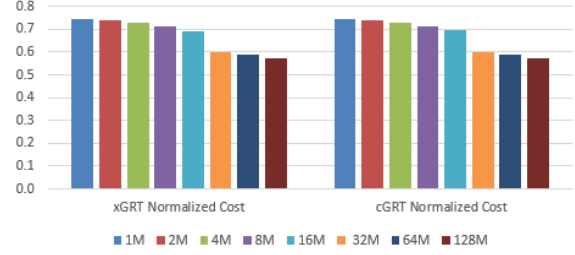
(c) GRT construction time (seconds), varying the number of bits (right)

Figure 2: Experiments on 1M rows

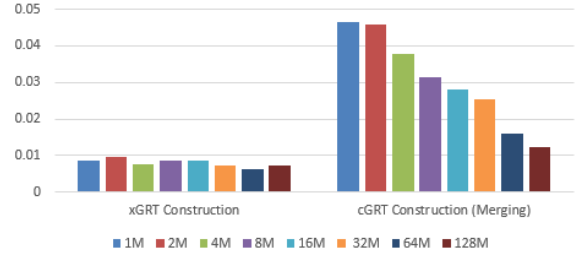
5 EXPERIMENTAL EVALUATIONS

We conducted experiments on a machine with Intel Core i7-4770 CPU. The dataset is from a real SAP application. We range-partitioned the table based on a date column, and extracted synopsis (MINMAX or GRT based) from the document identifier (SAP UUID) of each partition. The distribution of values in this column was sparse with many gaps. Fig. 2a compares GRT encoding with MINMAX based on the ratio of gaps preserved. For each column partition, the measure quantifies the ratio of values in column partition’s MINMAX range, that 1) do *not* appear in the column partition, and 2) can be excluded using the column synopsis. This ratio is between 0 and 1, with larger value more desirable; it indicates better pruning. This measure is zero for MINMAX; none of the values appearing in a column partition gap can be excluded by the minimum and the maximum range. The memory overhead for MINMAX for each partition is a pair of minimum and maximum values. For GRT based encoding, the memory overhead is one bit string per partition, plus one global range table for all partitions. Note that we need to have values (i.e. value pairs for MINMAX and value ranges in GRT) in memory, instead of dictionary value identifiers. This is mainly because the pruning is performed without accessing each column and its data structures (i.e. encoded data vector and dictionary).

Using GRT based encoding (Sec. 4.2.2), one can preserve more gaps as “zero” bits of the synopses to demonstrate values in the gap that are not included in a partition. The number of gaps preserved decreases when the number of bits dedicated to each synopsis reduces; merging GRT ranges produce wider the ranges in the target GRT table. This reduces the number of zeros as well as pruning opportunity. Fig. 2b reports the normalized GRT cost for no merge (xGRT of Sec. 4.2.1) and cGRT (Sec. 4.2.2). With reduction in the number of bits, the overhead of merge is observed as increase in the normalized GRT cost and in the construction time (Fig. 2c). For both xGRT and merged GRT, the normalized GRT cost decreases when the dataset size increases (Fig. 3a,b). The reason is that partitions become larger and the number or the size of gaps reduces in each partition. Therefore, the number of split ranges in xGRT decreases. Consequently, the time spent to merge these ranges reduces.



(a) Normalized GRT cost, varying the number or rows



(b) GRT construction time (seconds), varying the number or rows

Figure 3: Scalability

6 CONCLUSION AND FUTURE WORK

We introduced GRT, a compact data structure to achieve partition pruning. GRT is a list of ranges which we use to encode partitions and predicates. Bit string encoding is compact (compared to original partitions) and can be used for efficient partition elimination. This encoding is superior to MINMAX zone maps. In particular, when the distribution of values in column partitions is sparse and value gaps appear, pruning by MINMAX becomes less effective; false positives demand for extra candidate check [7]. Encoding partitions using global range tables, as opposed to local range tables optimized per partition (e.g. [9]), has the advantage that partition elimination can be performed without re-encoding the query per partition. Our results confirm the effectiveness of our approach, in preserving gap information. We studied the application of GRT in the context of query processing on cold partitions. However, its application can be extended to other use cases, e.g. semi-join reduction and enforcing the uniqueness constraint. Our encoding is reminiscent of Bloom filters [2]; we set bits using a global range table common to all partitions, to assist point and range queries. Designing GRT to offer bounds on false positive rates is an interesting future research direction. We plan to assign different sub-partitioning quota for dense vs. sparse partitions (non-homogeneous sub-partitioning). Sparse partitions should receive more encoding space than dense and uniform partitions, to reduce false positive rates.

REFERENCES

- [1] K. Alexiou and D. Kossmann and P. Larson. 2013. Adaptive Range Filters for Cold Data: Avoiding Trips to Siberia. *PVLDB* 6, 14 (2013), 1714–1725.
- [2] B. H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (1970), 422–426.
- [3] J. Kleinberg and E. Tardos. 2005. *Algorithm Design*.
- [4] N. Koudas. 2000. Space Efficient Bitmap Indexing. In *CIKM*.
- [5] G. Moerkotte. 1998. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *VLDB*.
- [6] A. Nica, R. Sherkat, M. Andrei, X. Chen, M. Heidel, C. Bensberg, and H. Gerwens. 2017. Statisticum: Data Statistics Management in SAP HANA. *PVLDB* 10, 12 (2017), 1658–1669.
- [7] D. Rotem, K. Stockinger, and K. Wu. 2005. Optimizing Candidate Check Costs for Bitmap Indices. In *CIKM*.
- [8] L. Sun, M. J. Franklin, S. Krishnan, and R. S. Xin. 2014. Fine-grained Partitioning for Aggressive Data Skipping. In *ACM SIGMOD*.
- [9] L. Sun, M. J. Franklin, J. Wang, and E. Wu. 2016. Skipping-oriented Partitioning for Columnar Layouts. *PVLDB* 10, 4 (2016), 421–432.

NoFTL-KV: Tackling Write-Amplification on KV-Stores with Native Storage Management*

Tobias Vinçon
DXC Technology
DBLab, Reutlingen University
Reutlingen, Germany
tobias.vincon@dxc.com

Sergey Hardock
DVS, TU-Darmstadt
Darmstadt, Germany
hardock@dvs.tu-darmstadt.de

Christian Riegger
DBLab, Reutlingen University
Reutlingen, Germany
christian.riegger@reutlingen-university.de

Julian Oppermann
ESA, TU Darmstadt
Darmstadt, Germany
oppermann@esa.tu-darmstadt.de

Andreas Koch
ESA, TU Darmstadt
Darmstadt, Germany
koch@esa.tu-darmstadt.de

Iliia Petrov
DBLab, Reutlingen University
Reutlingen, Germany
ilia.petrov@reutlingen-university.de

ABSTRACT

Modern persistent Key/Value stores are designed to meet the demand for high transactional throughput and high data-ingestion rates. Still, they rely on backwards-compatible storage stack and abstractions to ease space management, foster seamless proliferation and system integration. Their dependence on the traditional I/O stack has negative impact on performance, causes unacceptably high write-amplification, and limits the storage longevity.

In the present paper we present NoFTL-KV, an approach that results in a lean I/O stack, integrating physical storage management natively in the Key/Value store. NoFTL-KV eliminates backwards compatibility, allowing the Key/Value store to directly consume the characteristics of modern storage technologies. NoFTL-KV is implemented under RocksDB. The performance evaluation under LinkBench shows that NoFTL-KV improves transactional throughput by 33%, while response times improve up to 2.3x. Furthermore, NoFTL-KV reduces write-amplification 19x and improves storage longevity by imately the same factor.

1 INTRODUCTION

Over the last decade, various specialized DBMSs have been intensively investigated to meet the demand of new workloads, applications or data models. Persistent Key/Value stores (KV-stores) are specialized for high-throughput and predominantly update-intensive, OLTP-style workloads.

KV-stores exhibit a characteristic *lightweight architecture*, simplifying the deployment and integration process for large infrastructures and lowering maintenance demand in production. *Scalability* is intrinsically supported in terms of partitioning and distribution schemes, making KV-stores an excellent choice for current data-center architectures. The *simplicity* of their interface (with *put* and *get*) as well as data model matches wide range of modern insert and update intensive applications running high-throughput OLTP-style workloads. Last but not least, the ability to *serve as DB-Engines* in traditional and modern NoSQL databases (e.g. MyRocks[13] or MongoRocks), allows for the *integration* as meta stores into applications and distributed file systems (e.g. Ceph[10]), or serve as a backend for OLTP services.

*Produces the permission block, and copyright information

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

Persistent KV-stores leverage the properties of modern hardware due to the lean architecture, interface and flexibility, yet native hardware support is rare. The majority of such KV-stores rely on backwards-compatible storage, to ease administration and foster proliferation. Furthermore, the use of file systems simplifies space management, support for various storage architectures and the embedding in existing data center environments. *The underlying assumptions are that: (1) files and file-based I/O are the appropriate storage abstractions, and (2) use of standard/compatibility interfaces (and abstractions) on each individual layer of the I/O stack does not harm performance.*

The traditional I/O stack was developed with the characteristics of HDDs in mind, with the block-device interface, block I/O operations and files as abstractions. New storage technologies such as Non-Volatile Memories or Flash exhibit very different characteristics. However, to utilize them, persistent KV-stores require multiple layers of backwards compatibility, having a negative impact on performance and longevity. (1) Hardware resources are not fully exploited because of the hardware-oblivious abstractions. (2) DBMS access patterns result in suboptimal physical I/O patterns due to the presence of multiple abstraction layers along the critical I/O path. (3) KV-store information about the current workload cannot be used for better physical data placement. (4) Functionality along this critical I/O path is redundant. Significant write-amplification and suboptimal performance are the inevitable consequences.

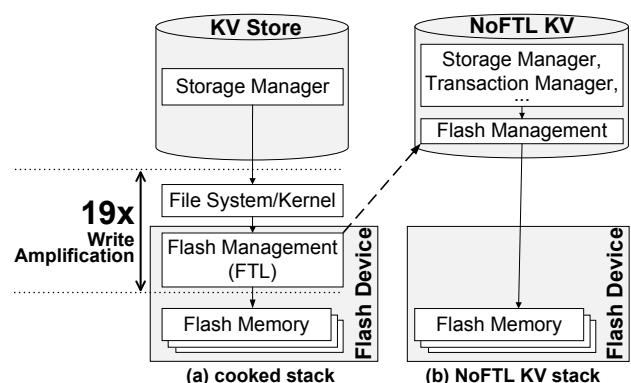


Figure 1: Write-Amplification along a traditional I/O stack in contrast to NoFTL-KV.

To verify the above claims we perform an experiment under RocksDB running LinkBench and measure the end-to-end write-amplification along a backwards-compatible, file-system based stack. The results (Fig. 2a) indicate a 19x physical write volume increase, lower performance and longevity.

In this paper we present NoFTL-KV (Fig. 1), an approach that avoids backwards compatibility and targets the above disadvantages by controlling the underlying physical storage directly. NoFTL-KV integrates physical storage (Flash) management natively in the KV-store. Subsequently, it opens up ways for workload adaptability within the storage layer and new abstractions for native storage.

The main contributions of this paper are:

- (1) The extension of the concept of native storage management (NoFTL) to persistent KV-stores. We show that by coherently integrating address mapping, data placement, GC and free space management into the KV-store, storage characteristics, on-device parallelism and wear-leveling are addressed.
- (2) NoFTL-KV is implemented under RocksDB.
- (3) The performance evaluation under LinkBench[1] shows that NoFTL-KV improves the transactional throughput by 33%, while the response times improve up to 2.3x. Furthermore, NoFTL-KV improves physical storage management. In terms of *write efficiency*, NoFTL-KV performs 87% less physical page writes (including maintenance I/O and GC). Moreover, NoFTL-KV performs 19x less erases, improving the endurance by approximately the same factor.

The rest of the paper is structured as follows. NoFTL-KV and the integration into RocksDB are described in Section 3. Experimental results are discussed in Section 4. We conclude in Section 5.

2 RELATED WORK

Modern workloads (Social Media, Big Data or IoT) not only have become write-intensive and require high sequential throughput, but also demand low latencies [16]. *Read- and Write-Amplification* are major performance factors [16].

These can either be approached by utilising compression to decrease I/O in general [5] or by aligning better with the characteristics of modern storage devices. The latter is addressed in terms of either new data structures [3, 4, 19], or new software interfaces [2] as well as Flash interface extensions [9, 12, 14]. However, neither of those takes the issues with the *cooked stack* into account. [6] and [7] present a full integration of native storage support within traditional DBMS. A few lightweight KV-stores address the concept of direct native storage integration [8, 15, 17, 18] by moving the entire KV-store onto the device. Yet, physical storage management is only partially addressed.

With NoFTL-KV we address the deep integration of native storage management to tackle all issues regarding the traditional cooked stack while avoiding to overload the device controller with database functionality and maintaining a mature KV-store.

3 NOFTL-KV: NATIVE STORAGE KV-STORE

We investigate the concept of native storage management and NoFTL under persistent KV-stores to address and evaluate the above mentioned claims. RocksDB exhibits an append-only I/O pattern for various write-intensive workloads, because of its LSM-Tree-based persistent storage. LSM-trees perform regular compactions to remove old records, to ensure optimal tree structure and to perform hot-cold-separation. Compactions reorganize

levels of the LSM-Tree, removing updated or deleted KV-Pairs, at regular intervals or given a certain threshold. As a consequence, frequently changing data is placed in the upper levels of the LSM-Tree, while the lower levels contain the cold data.

Under NoFTL-KV we pursue coherent integration of Flash management into existing modules of the KV-store as shown in Fig. 3. Firstly, NoFTL-KV has direct control over hardware resources through a native storage interface (NSI). NSI allows the DBMS to operate with I/O operations, in granularity and with addressing schemes supported by the underlying storage technology. Furthermore, NSI eliminates the need to support backwards compatibility. Secondly, we revisit hardware-oblivious abstractions and propose using physical storage abstractions such as *Regions* to: (a) reduce read/write amplification along the I/O path, (b) utilize available I/O parallelism more efficiently, (c) provide better hot-cold data separation to (d) improve space management and (e) increase longevity.

Moreover, unnecessary DBMS data transfers can be reduced by pushing tasks down to the storage device. For instance, parts of garbage collection and compaction can be planned by the NoFTL-KV storage manager for certain Regions, but are executed onto the device to reduce I/O contention and data transfers. Likewise, queries, i.e scans, can be pushed down and executed on the storage device. Especially in combination with Regions, such queries can profit by the involved address mapping and level of on-device-parallelism. Also worth to mention is that processors on such storage devices usually exhibits the characteristics of common co-processor (ASIC or FPGA). These are perfectly aligned to the characteristics of modern storage technology (Flash, NVM) e.g. in respect to parallelism.

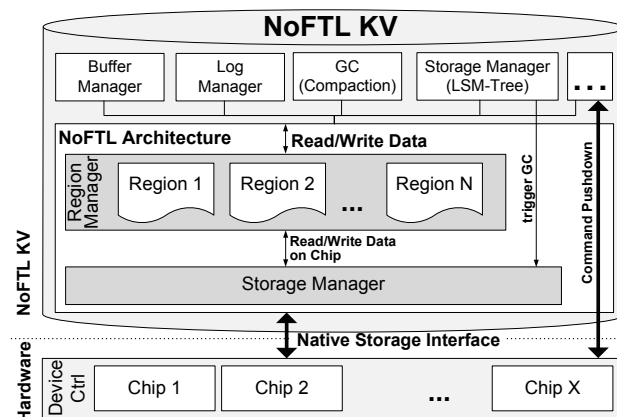
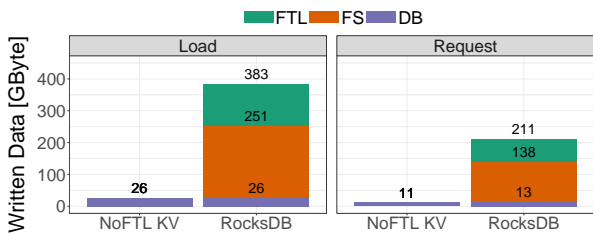


Figure 3: NoFTL-KV: Design of a deep integration of the NoFTL concept within an entire KV-store for native storage management

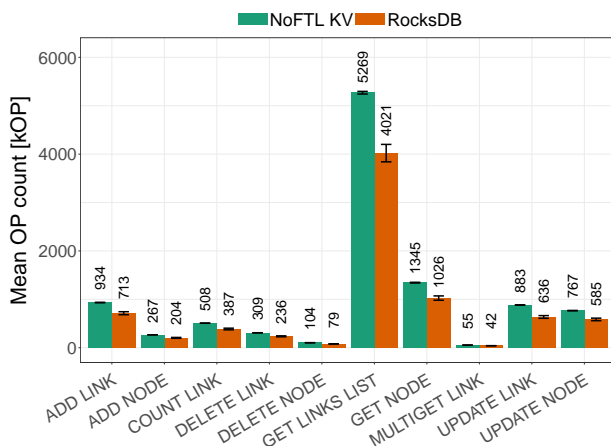
By integrating address mapping into the storage manager of the KV-store, the latter gets control over the physical data placement on Flash. Hence, the KV-store can utilize available information about data semantics, statistics and the access pattern (e.g., desired level of I/O parallelism) to perform efficient placement. Individual levels of the LSM-Tree can be physically separated on different chips to improve I/O throughput and parallelism since I/O-heavy compaction jobs do not block the entire device. Consequently, new storage abstractions can be defined besides files.



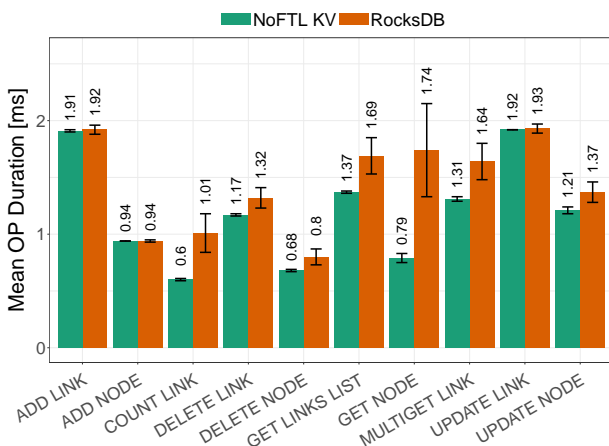
(a) Amount of data written by the DB, FS and FTL during the load and request phases of LinkBench shows the write-amplification of RocksDB in contrast to NoFTL-KV

	NoFTL-KV		RocksDB		Speedup
	Mean	StDev	Mean	StDev	
GC Calls	1	3	4769	2434	
GC Page Write	0	0	1932263	737453	
Block Erase	1127	3564	21389	7792	18.98x

(b) Number of logical page writes of the DB's compaction and physical page writes of the device over 10 request phases demonstrates write-amplification and inconsistent logical to physical page write ratio.



(c) Throughput: The average number of executed operations over the last 7 request phases demonstrates that NoFTL-KV outperforms RocksDB



(d) Response Time (the lower the better): Average operation latencies and std. dev. are better and more stable under NoFTL-KV vs. RocksDB

Figure 2: Results of the experimental evaluation of NoFTL-KV using LinkBench

We introduce *Regions* as physical storage abstractions spanning multiple chips/dies (i.e. parallel unit of the storage device). They can be effectively optimized for different access patterns (sequential, random, append) of various KV-store components (e.g. Levels of LSM-Tree and Log Manager). Regions allow for flexible physical storage management as the parameters of hot-cold data separation, garbage collection etc. are part of the definition. The level of supported I/O parallelism per Region can be defined in terms of the number of chips/dies it spans or whether these run in pseudo SLC, MLC or TLC Flash mode. Region definitions are not static, but can evolve over time to reflect properties of the workload.

```
CREATE REGION rgBlockMapping (
  MAX_CHIPS=4, MAX_CHANNELS=4, ..., ADDR_MAPPING=BLOCK,
  NAND_MODE=MLC, ...);
CREATE TABLESPACE MyRocks.tb1Block (
  REGION=rgBlockMapping, UNIFORM EXTENT SIZE 128K );
CREATE TABLE MyRocks.nodetable(...)
  TABLESPACE MyRocks.tb1Block;
```

Furthermore, the functional redundancy along the *cooked I/O stack* is reduced. While, the file-system and the FTL distort the append-based access pattern and amplify the read/write data volume, NoFTL-KV simplifies the critical I/O path, exhibits a physically sequential I/O pattern, and offers better physical storage management. Consequently, write-amplification is significantly reduced. Similarly, the integration of the garbage collection within the compaction process of the LSM-Tree, allows for

elimination of the time and resource-expensive merges common for traditional FTL-based SSDs. As a result, the KV-store is able to trigger the GC only when necessary and under the current workload. Higher longevity through less block erases and better throughput are the consequence.

4 EXPERIMENTAL EVALUATION

Testbed. Our testbed comprises a server equipped with an Intel Xeon E5-1620 v3 3.50 GHz CPU-core, 32GB RAM and an Intel DC 3600 SSD under Ubuntu 12.04 LTS, kernel 3.13.0.

The Flash storage device for the NoFTL-KV data is emulated by our real-time Flash Simulator[6], which is running as a kernel module. Configured with common latencies for reads, writes, and erases of current SLC NAND Flash it is able to simulate a modern enterprise SSD with either block- or char-device interfaces. For the block device, FASTER[11] is utilised as FTL with an over-provisioning area of 14%. In our setup, the simulator consumes 24 GB of memory to emulate an SSD of the same capacity with 256 pages (4KB) on 24576 blocks. The level of parallelism (emulated NAND chips/dies) is limited by the number of hardware threads. For our experiments we configured NoFTL-KV to only store RocksDB LSM-Tree files on the emulated device and the remaining files on the Intel DC 3600 formatted with an ext4 file system.

LinkBench. The experimental evaluation is performed using LinkBench[1], which is an OLTP-style workload on large updatable graphs. Under LinkBench the working data set size is an order of magnitude larger than the database buffer. The request phase of LinkBench comprises common graph queries like adding, getting, counting, deleting, updating nodes or edges on the graph. The experimental dataset is a graph of 15M nodes (initial), amounting to 15GB raw data. The number of requests and duration vary depending on the experiment. The baseline utilises the same configuration (ext3, 4KB blocksize, active journal) for MyRocks with RocksDB, hereinafter referred to as RocksDB.

Write-Amplification. To measure write amplification, hooks are placed in the storage engine of RocksDB (DB), the file system (FS), and the Flash emulator (FTL), i.e., in all layers along the I/O path. The number of requests is limited to 1M per thread with sufficient time (10h) to be executed completely. This ensures that, at the end, both variants have executed the same number of operations. The results (Fig. 2a) for NoFTL-KV and the baseline RocksDB represent average values of multiple runs.

Not surprisingly, the significant write amplification of the cooked stack becomes evident. The 26 GB of raw data, bulk-loaded during the *load phase*, swells up by more than 14 times to 383 GB. On top of that, the file system adds about 225 GB and the FTL increases this again by 132 GB. During the *request phase*, the disadvantages of the cooked I/O stack become even more visible. The average write-amplification here is more than 19x. This creates enormous I/O overhead, which is clearly reflected by the metrics to follow.

Throughput. The mean number of executed operations and their errors for every operation type is shown in Fig. 2c. NoFTL-KV outperforms RocksDB in every type of query. This is because of the smaller data volume to be written, and the better utilisation of available Flash parallelism. The workload of LinkBench has a high write-intensity over the complete duration. Consequently, the throughput increases about 31% across all operation types. The performance stability across different runs, indicated by the error bars increases by an order of magnitude.

Response Time. To investigate the impact on response time for common operations, we perform further experiments with 1M requests per thread. Fig. 2d shows the average duration, while the error bars indicate the standard deviation.

One can clearly see that the latency is lower under NoFTL-KV. Especially reading operations like *GetNode()*, *GetLinksList()*, and *MultigetLink()* perform significantly better. This is even more relevant, since about 22% of these could not be served by database buffer (cache miss rate) and are read from the persistent device. On the other hand, inserting and updating operations like *AddNode()*, *AddLink()*, and *UpdateLink()* complete directly after pushing the data into an in-memory buffer and the WAL. This buffer is persisted only after a compaction, which is not taken into account for the operation latency. This explains the similar performance for both RocksDB and NoFTL-KV. The only exception is *UpdateNode()*, which might result in multiple gets that are slower with the traditional I/O stack.

Erases and Longevity. Write-amplification on Flash devices inevitably leads to more physical Flash erases, which has negative impact on device longevity. Table 2b captures the GC activity in terms of physical page writes and block erases during the benchmark runs of the previous experiment.

RocksDB performs almost 19 times the physical block erases than NoFTL-KV, which is primarily due to (i) the journal of the file system, which doubles Flash page writes, and (ii) FASTER's

hybrid address mapping scheme. It is worth noting that the erase overhead of FASTER would also be present in other FTLs, which utilize hybrid address translation (common for current SSDs). As soon as the so-called log block area of the device runs out of space, the GC kicks in and merges the updated data with the corresponding Flash blocks in the data block area. Each of those merges requires multiple page migrations (on-device write amplification), and one or two erase operations (partial or full merges). NoFTL-KV is configured to use BLM, which matches the append-only LSM-Tree based storage management of RocksDB.

5 CONCLUSION

In the present paper we propose NoFTL-KV, an approach that results in a lean I/O stack, integrating physical storage management natively in the Key/Value store. NoFTL-KV eliminates backwards compatibility, allowing the Key/Value store to directly exploit the characteristics of modern storage technologies. NoFTL-KV is implemented under RocksDB and evaluated using LinkBench. The transactional throughput improves by 33%, while response times improve up to 2.3x. Furthermore, NoFTL-KV reduces write-amplification 19x and improves endurance. In addition, our current integration on the file-based LSM-Tree can be further improved by a deeper integration into the KV-store's data structure in future work to gain additional performance improvements.

REFERENCES

- [1] Timothy G. Armstrong, Vamsi Ponnkanti, Dhruva Borthakur, and Mark Callaghan. LinkBench: A Database Benchmark Based on the Facebook Social Graph. In *Proc. SIGMOD 2013*.
- [2] Matias Björling, Javier Gonzalez, and Philippe Bonnet. LightNVM: The Linux Open-Channel SSD Subsystem. In *Proc. FAST 2017*.
- [3] Niv Dayan, Philippe Bonnet, and Stratos Idreos. GeckoFTL: Scalable Flash Translation Techniques For Very Large Flash Devices. In *Proc. SIGMOD 2016*.
- [4] Biplob Debnath, Sudipta Sengupta, and Jin Li. SkimpyStash: RAM Space Skimpy Key-value Store on Flash-based Storage. In *Proc. SIGMOD 2011*.
- [5] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruva Borthakur, Tony Savor, and Michael Strum. Optimizing Space Amplification in RocksDB. In *Proc. CIDR 2017*.
- [6] Sergej Hardock, Ilia Petrov, Robert Gottstein, and Alejandro Buchmann. NoFTL: Database Systems on FTL-less Flash Storage. In *Proc. VLDB 2013*.
- [7] Sergey Hardock, Ilia Petrov, Robert Gottstein, and Alejandro P. Buchmann. NoFTL for Real: Databases on Real Native Flash Storage. In *Proc. EDBT 2015*.
- [8] Y. Jin, H. W. Tseng, Y. Papakonstantinou, and S. Swanson. KAML: A Flexible, High-Performance Key-Value SSD. In *In Proc. HPCA 2017*.
- [9] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Gi-Hwan Oh, and Changwoo Min. X-FTL: Transactional FTL for SQLite Databases. In *Proc. SIGMOD 2013*.
- [10] Dong-Yun Lee, Kisik Jeong, Sang-Hoon Han, Jin-Soo Kim, Joo-Young Hwang, and Sangyeun Cho. Understanding Write Behaviors of Storage Backends in Ceph Object Store. In *Proc. MSST 2017*.
- [11] S. P. Lim, S. W. Lee, and B. Moon. FASTER FTL for Enterprise-Class Flash Memory SSDs. In *Proc. SNAPI 2010*.
- [12] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, and Raju Rangaswami. NVMKV: A Scalable, Lightweight, FTL-aware Key-value Store. In *Proc. ATC 2015*.
- [13] Yoshinori Matsunobu. InnoDB to MyRocks Migration in Main MySQL Database at Facebook. In *Proc. SREcon 2017*.
- [14] Gihwan Oh, Chiyong Seo, Ravi Mayuram, Yang-Suk Kee, and Sang-Won Lee. SHARE Interface in Flash Storage for Relational and NoSQL Databases. In *Proc. SIGMOD 2016*.
- [15] Samsung. http://www.samsung.com/semiconductor/global/file/insight/2017/08/Samsung_Key_Value_SSD_enables_High_Performance_Scaling-0.pdf
- [16] Russell Sears and Raghu Ramakrishnan. bLSM: A General Purpose Log Structured Merge Tree. In *Proc. SIGMOD 2012*.
- [17] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A User-Programmable SSD. *Proc. OSDI 2014*.
- [18] Shuotao Xu, Sungjin Lee, Sang-Woo Jun, Ming Liu, Jamey Hicks, and Arvind. Bluecache: A Scalable Distributed Flash-based Key-value Store. In *Proc. VLDB 2016*.
- [19] Jiacheng Zhang, Jiwu Shu, and Youyou Lu. ParaFS: A Log-structured File System to Exploit the Internal Parallelism of Flash Devices. In *Proc. ATC 2016*.

Towards Hypothetical Reasoning Using Distributed Provenance

Daniel Deutch, Yuval Moskovitch, Itay Polack Gadassi and Noam Rinetzky
Tel Aviv University

ABSTRACT

Hypothetical reasoning is the iterative examination of the effect of modifications to the data on the result of some computation or data analysis query. This kind of reasoning is commonly performed by data scientists to gain insights. Previous work has indicated that fine-grained data provenance can be instrumental for the efficient performance of hypothetical reasoning: instead of a costly re-execution of the underlying application, one may assign values to a pre-computed provenance expression. However, current techniques for fine-grained provenance tracking are ill-suited for large-scale data due to the overhead they entail on both execution time and memory consumption.

We outline an approach for hypothetical reasoning for large-scale data. Our key insights are: (i) tracking only relevant parts of the provenance based on an a priori specification of classes of hypothetical scenarios that are of interest and (ii) the distributed tracking of provenance tailored to fit distributed data processing frameworks such as Apache Spark. We also discuss the challenges in both respects and our initial directions for addressing them.

1 INTRODUCTION

Data analytics often involves *hypothetical reasoning*; repeatedly modifying the database according to specific scenarios, and observing the effect of such modifications on the result of some computation. A naive way to perform such an analysis is to create a copy of the data, modify it, and recompute the results for the inspected scenario. However, this approach can be very costly when the computation involves access to large-scale data. A more efficient method is to use *provisioning* [4, 6]: compute a symbolic *provenance expression* (PE) which encodes the result of the computation under any possible scenario. Then, the user interacts with the PE for efficient exploration of the scenarios. Creating the PE incurs a *one-time* overhead over the evaluation of a specific query. However, if the analyst inspects multiple scenarios, the creation of the PE may pay off in an inspection which is orders-of-magnitude more efficient than a naive recomputation.

Example 1.1 (Running example). Consider a database of a telephony company containing the number, name, zip code, and call plan of every customer, the *price per minute* (*ppm*) of every plan, and a log of the duration and date of every call (see Fig. 1). The following query computes the company revenues (this example is inspired by the one used in [6]):

```
SELECT Calls.Mo, SUM(Calls.Dur * Plans.Price)
FROM Calls, Cust, Plans
WHERE Cust.Plan = Plans.Plan
      AND Cust.Num = Calls.Num
GROUP BY Calls.Mo
```

The query computes the monthly revenues by summing the per-call-revenue, computed by multiplying the duration of every call

Cust				Plans	
Num	Name	Zip	Plan	Plan	Price
555-777	Bob	10001	Plan1	Plan1	0.1
555-942	Alice	10002	Plan2	Plan2	0.2
555-465	Dave	10003	Plan2		

Calls					
Num	Mo	Dur	Num	Mo	Dur
555-777	Jan	21	555-942	Feb	33
555-777	Jan	8	555-942	Feb	27
555-777	Jan	14	555-942	Feb	32
555-777	Feb	7	555-942	Feb	16
555-777	Feb	17	555-465	Jan	9
555-777	Feb	33	555-465	Jan	7
555-942	Jan	28	555-465	Jan	8
555-942	Jan	20	555-465	Feb	33
555-942	Jan	23	555-465	Feb	14
555-942	Jan	21	555-465	Feb	12

Figure 1: Example database

by the *ppm* of the customer’s plan and grouped by month. An analyst may be interested in the effect of possible changes to the call price on the company revenues. For example, the analyst may wish to compute the revenues under (some combination of) the following hypothetical scenarios:

- HS1 What if the *ppm* is decreased by 10% in calling plan 1 and set to \$0.3 in calling plan 2?
- HS2 What if all the customers with vanity phone numbers were subscribed to plan 1?
- HS3 What if the *ppm* for costumers in Boston is set to \$0.3?
- HS4 What if a 25% discount is given to calls which took less than ten minutes?

Computing a PE for the query of Example 1.1 and the scenario HS1 is relatively straightforward for small-scale data [4, 6] (see Section 2.1). Scenario HS2 is more challenging, since it involves changes to the parts of data that is in a comparison in the query. This may be handled in a c-table-like construction as mentioned in [6] (see Section 2.2). The two other scenarios involve changes that may not be directly applied to input data, but rather only to some views over it; we discuss the semantics of such scenarios and the questions that arise in Section 2.3. Further, for all scenarios, there is the concern of space and time overhead incurred by provenance tracking. We propose distributed provenance representations to address this, in Section 3. Specifically, we present two different methods for representing fine-grained provenance in a distributed manner: a *combined representation* that stores each polynomial as a single tuple in the output table comprised of a collection of monomials, and a *separated representation* that stores separately each monomial. Our preliminary results, based on an implementation in *Apache Spark* [12, 14, 15], indicate that the combined approach allows for more efficient exploration, provided that the provenance expression is relatively small, while the separated approach scales better as it is more suitable to the

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

distributed model, however it incurs a non-negligible overhead over the combined approach where the latter can be used.

Related Work. Data provenance has been studied for different data transformation languages, from relational algebra to Nested Relational Calculus, with different provenance models and applications and with different means for efficient storage. Particularly relevant are systems that support provenance for distributed systems [3, 5, 9, 11, 13]. We note that the Spark framework already supports a provenance-like logging mechanism that allows for fault tolerance. Unlike those existing works, we focus on cell-based provenance, where polynomials replace individual values, which is necessary for answering what-if questions. The use of data provenance for hypothetical reasoning has also been studied in e.g., [4, 6, 7], but without targeting big data and addressing the scalability issues that consequently arise.

2 PROVISIONING IN A NUTSHELL

Provisioning, i.e., *hypothetical reasoning* using provenance, is a two step process: First, the analyst *parameterizes* some entries in the input database. Then, she explores the effect of hypothetical modifications by instantiating the parametrized entries with specific values [4, 6]. Technically, parametrization entails instrumenting entries using *symbolic variables*, e.g., by (symbolically) adding or multiplying them with the original numerical values. When the provisioning engine runs an SQL query, it treats the parameterized entries in a symbolic way, and creates a *provenance expression* (PE); an *output table* which may contain symbolic values. The symbolic representation allows the analyst to explore multiple scenarios by *evaluating* the PE using a specific *assignments of concrete values* to symbolic variables. Different types of scenarios lead to different challenges as we next explain.

2.1 Basic Provisioning

We start with the case where the parametrization involves a single table and the program does not perform selection or join over parameterized attributes. In this case, the semantics of provisioning is fairly straightforward. First, the parameterized data entries are annotated with variables which, roughly speaking, replace every parameterized data by a *multivariate polynomial*. Second, a query Q is executed according to its standard semantics except that summations and multiplications are executed symbolically to produce polynomials instead of concrete values. As for aggregation, we simply combine the polynomials in the provisioned attributes using a symbolic plus (+) operator. The analyst uses the generated provenance expression to explore a specific scenario by providing an *assignment* ρ which defines a (concrete) valuation for each symbolic variable and computing the value of the polynomial under ρ . It is straightforward to observe that this computation produces the same results as the re-execution of Q on a correspondingly modified database.

Example 2.1. Reconsider our running example query and scenario HS1 from the Introduction. To generate a PE for them (where we wish to support a generalized version of HS1), we first parameterize the input database as shown in the Plans table of Figure 3. In this example, the price for plan 1 is multiplied by a variable p_1 and the price for plan 2 is both multiplied by a variable p_2 and added a variable s_2 , the latter allowing for its replacement by a different value. Evaluating the running example query using these symbolic values results in the output table and provisioning expressions shown in Figure 2.

Provenance expression		Output	
Mo	Revenues	Mo	Rev.
Jan	$43 \cdot (0.1 \cdot p_1) + 116 \cdot (0.2 \cdot p_2 + s_2)$	Jan	38.67
Feb	$57 \cdot (0.1 \cdot p_1) + 167 \cdot (0.2 \cdot p_2 + s_2)$	Feb	55.23

Figure 2: A PE and the results of its evaluation according to scenario HS1 (i.e., $\rho = [p_1 \mapsto 0.9, p_2 \mapsto 0, s_2 \mapsto 0.3]$)

2.2 Conditional Provisioning

Hypothetical scenario HS2 in Example 1.1 is a particular member in a family of scenarios which enable computing the company’s monthly revenues under different reassignments of customers to plans. A PE which allows inspecting the scenarios in this family can be generated by the techniques of [4, 6]. In our example, this entails parameterizing the Plan attribute in table Cust by replacing its content in every entry by a customer-unique symbolic variable, and then representing the join of Cust and Plans over the parameterized attribute using a c-table [10].

Example 2.2. Reconsider Figure 3, and now note that the Cust table is also parameterized, adding the parameters n_{777} , n_{942} and n_{465} for HS2. Each of these may be assigned to Plan1 or Plan2. The resulting PE for January in this example is

$$43 \cdot (0.1 \cdot p_1) \cdot [n_{777} = \text{Plan1}] + 43 \cdot (0.2 \cdot p_2 + s_2) \cdot [n_{777} = \text{Plan2}] + 92 \cdot (0.1 \cdot p_1) \cdot [n_{942} = \text{Plan1}] + 92 \cdot (0.2 \cdot p_2 + s_2) \cdot [n_{942} = \text{Plan2}] + 24 \cdot (0.1 \cdot p_1) \cdot [n_{465} = \text{Plan1}] + 24 \cdot (0.2 \cdot p_2 + s_2) \cdot [n_{465} = \text{Plan2}]$$

Expressions in brackets are mapped to 1 or 0 based on their truth value upon assignment of values to the variables. If we are still interested in exploring HS1, we simply assign to the n variables their original values, and proceed to assigning the other variables as in the previous example. But we can also, e.g., assign Plan2 to n_{777} , intuitively switching the plan of Bob to be Plan2; then the expression $[n_{777} = \text{Plan1}]$ is evaluated to 0 and $[n_{777} = \text{Plan2}]$ to 1, so that Bob is given the price of Plan2.

2.3 View-Based Provisioning

The parameterization of the database in Examples 2.1 and 2.2 is done over a single table for each hypothetical scenario. However, the symbolic representation needed for capturing scenarios HS3 and HS4 require the pre-generated views produced by joining tables Cust and Plans and tables Cust and Calls, respectively. The view is required because Price is parameterized based on the customer’s Zip code in HS3 and the call’s duration (Dur) in HS4. We can parameterize a view as if it is a single (albeit joined) table, and use it to generate a provenance expression as described in Section 2.1. However, the use of a parameterized view may render some queries as undefined for hypothetical reasoning as the following example shows.

Example 2.3. Retrieving the *ppm* of Plan1, using the parameterized view generated for scenario HS3 does not make sense because the price of the plan depends on the customer’s zip code. Conversely, determining the monthly revenues over this view is

Cust				Plans	
Num	Name	Zip	Plan	Plan	Price
555-777	Bob	10001	n_{777}	Plan1	$0.1 \cdot p_1$
555-942	Alice	10002	n_{942}	Plan2	$0.2 \cdot p_2 + s_2$
555-465	Dave	10003	n_{465}		

Figure 3: Data with provenance annotation

well defined, and can be computed using, e.g., a rewrite of the query shown in Example 2.1.

The problem of using views for hypothetical reasoning is highly related to the problem of answering queries using views [8]. One notable difference between the two problems is that a query may be well defined under an hypothetical scenario also if it only relies on tables that are *not* used in the views; the problem arises only when the query uses the same pieces of data affected by the parameterization, but does so in a “different” way. Formalizing and studying the properties that make a query answerable under a definition of hypothetical scenarios over views is an intriguing problem for investigation.

3 DISTRIBUTED BASIC PROVISIONING

Provisioning large-scale data is challenging: computing and maintaining provenance expressions may lead to large overheads in terms of both memory and time. We propose a partial remedy to this problem via an adaptation of basic provisioning to the distributed setting, and report on initial promising experimental results obtained in the context of *Apache Spark* [12, 14, 15].

Simplifying assumptions. We focus on basic provisioning and further assume that only numerical attributes may be parameterized and that the parameterization is performed on each *record* separately—parameterization based on information located at multiple tables requires creating an appropriate (unparameterized) view at a preliminary stage.

3.1 Apache Spark

Apache Spark is a popular framework for writing large scale data processing applications. *Spark* provides operations such as *map*, *filter* and *fold* which can be seen as extensions to the standard database operations *project*, *select* and *aggregation*, respectively, with arbitrary UDFs applied. In addition, *Spark* provides a *natural join* operation between multisets indexed by a common (possibly non-unique) key and *foldByKey* operations which, analogously to the *groupByKey* operation in databases, aggregates together the values pertaining to the same key.

Spark programs are executed on a cluster comprised of a single *master* node, which coordinates one or more *worker* nodes. Roughly speaking, a *spark* program operates as follows: It first partitions the data across the cluster nodes. *map* and *filter* operations are executed by each node, and on each partition in parallel. *fold* operations are executed in two stages: Every worker node aggregates the values in its partitions and sends the partial result to the master node which aggregates them together. *foldByKey* and *join* operations may require a preliminary (expensive) *shuffle* stage where records are exchanged between working nodes according to a strategy determined by the driver. At the end of the shuffle stage, all the records pertaining to any key are located in the same partition. This allows to compute the (by-key) *fold* and *join* operations by every working node separately.

3.2 Distributed Provenance Expressions

Representing and interacting with a provenance expression can be computationally expensive when it is generated from large-scale data. Specifically, two challenges may arise: (i) the table may contain many polynomials, thus representing it would consume a lot of space, and (ii) every polynomial may be comprised of a large number of monomials, thus designing efficient ways to represent and evaluate large polynomials is required.

Key	Value
(Jan, p_1)	$43 \cdot 0.1$
(Jan, p_2)	$116 \cdot 0.2$
(Jan, s_2)	116
(Feb, p_1)	$57 \cdot 0.1$
(Feb, p_2)	$167 \cdot 0.2$
(Feb, s_2)	167

Figure 4: Separated provenance representation.

Handling challenge (i). Using a *polynomial-level distributed representation*, which we refer to as the *combined representation*. In this representation, the provenance expression is spread across the cluster so that different nodes manage different polynomials (and each node manages a whole polynomial). Technically, the *combined representation* stores every polynomial as an additional attribute of each relation. The attribute is an array of *monomials*, where every monomial is a pair comprised of a coefficient and an array of symbolic variables. The symbolic execution of an aggregation operation amounts to combining the monomials coming from different polynomials by performing algebraic simplifications to compute the coefficients. For example, the provenance expression shown in Figure 2 is in the combined representation. Evaluating the provenance expression is done by an invocation of a map operation that evaluates every polynomial for the given assignment in the standard way.

Handling challenge (ii). Using *monomial-level distributed representation*, which we refer to as the *separated representation*: We split every polynomial into its constituent monomials and store each monomial in its own record. Technically, every monomial is represented as a key-value pair, where the value is the monomial’s coefficient and the key is a combination of a unique identifier of the polynomial that the monomial belongs together with the product of its variables. For example, Figure 4 depicts a separated representation of the PE shown in Figure 2. Under the separated representation, the (symbolic) execution of an aggregation operation does not combine different symbolic values into a single polynomial. Instead, when evaluating the provenance expression, a map operation evaluates the value of every monomial and a *foldByKey* operation computes the value of every polynomial.

3.3 Experimental Evaluation

We have performed a preliminary evaluation of our approach, and show two basic experiments based on our running example and a synthetically generated database. In the first experiment, we have joined the *Cust* and *Plans* tables, and parameterized the *Price* attribute of every customer record by multiplying it with a unique customer-unique variable. We ran the query using a *Cust* table that contained 2^{18} (262,144) customers, identified by their unique phone number, and uniformly distributed across the 41,668 US zip codes, and eight call plans. This resulted in a provenance expression comprised of 1536 polynomials. We considered different sizes of the input data by populating the *Calls* table with either 128, 256, 512, 1024, or 2048 calls for every customer. The resulting sizes of the *Calls* tables are in the range of 500 MB to 9.1 GB.

Creating the provenance expressions was up to $\times 2.1$ slower than the computation of the non-provisioned query. Evaluating the queries, on the other hand, up to $\times 385$ time faster than a naive recomputation. Figure 5 depicts the provenance evaluation time

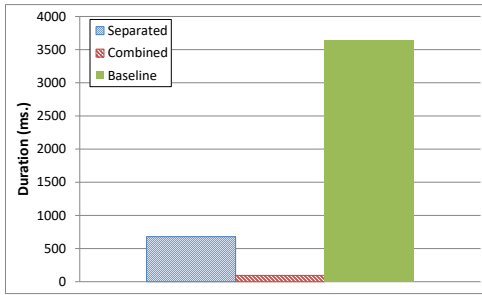


Figure 5: Experiment 1 (Assignment time)

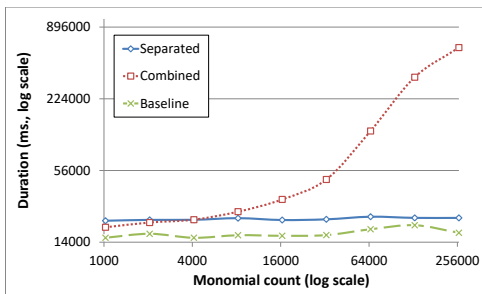


Figure 6: Experiment 2 (PE generation time)

for the *smallest* database¹. We made sure that changing the size of the data only affected the value of the coefficients, specifically, it did not change the number of monomials in any of the polynomials. As a result, the time it took to evaluate the provenance expression was independent of the size of the database.

In the second experiment, we evaluated the efficacy of the different representations using polynomial of different sizes by generating a provenance expression that comprised of a single polynomial with different number of monomials. We joined the Cust and Plans tables, and parameterized the Price attribute of every customer by multiplying it with a symbolic variable. We used the same database as in the first experiment using a Calls table with 32 calls for every customer. We ran the experiment nine times, where in the i th experiment, for $i = 0..8$, we used the same variable for all the customers whose phone numbers ended with the same $10 + i$ bits. Thus, the smallest polynomial contained 1024 monomials, and the biggest one had 262144. Figure 6 depicts the time it took to generate the provenance expressions. It shows that the Combined approach has the upper hand for small polynomials, yet it quickly becomes much slower than the Separated approach, up to the point of being prohibitively expensive.

Analysis. The Separated representation has an advantage for big polynomial whereas the Combined approach is better when the polynomials are small. We believe that the reason for this is that the Separated representation allows to store and evaluate a

¹We ran our experiments on Amazon EC2 public cloud [2] using a cluster comprised of nine m4.xlarge virtual machines. Eight machines were used as workers and one as the cluster’s driver. Each virtual machine has four virtual CPUs, 16GB memory. (The physical CPU used for an m4.xlarge virtual machine is a 2.4GHz Intel Xeon E5-2676 v3 processor, where every virtual CPU is a hyperthread of an Intel Xeon core.) We ran Spark version 1.6.2 over Amazon Linux AMI 2016.03, with Linux kernel 4.4.11, Java runtime 1.8.0_101, and Scala version 2.10.5. The UDFs were implemented in Scala, and the data was stored in Amazon Simple Storage Service (S3) [1].

single polynomial concurrently using multiple nodes. Furthermore, this representation is particularly beneficial in the context of Spark which handles tables containing a large number of records very efficiently, but struggles when it is asked to process large records due to its in-memory representation of the data and its single-threaded record processing.

4 OPEN PROBLEMS AND FUTURE WORK

We briefly discussed in this paper the problem of provisioning for big data, highlighting semantic and scalability issues, and proposing partial solutions in the context of Apache Spark. Both facets of the problem require extensive investigation, which is the subject of our on-going work. On the semantic side, a notable omission in the current literature is a formal language for specifying hypothetical scenarios (the need for such language is also mentioned in [4]); then, given a formal specification, can we efficiently decide whether an hypothetical is “answerable” (as in the view-based examples we have shown, this is not a given) and if so efficiently generate the PE? In terms of efficiency, we plan to extend our Spark-based implementation to allow for conditional provisioning. In addition, we are interested in exploring the benefits of a *hybrid* method for representing distributed provenance which combines the methods Separated and Combined we have presented by maintaining every polynomial as a table of sub-polynomial instead of single monomials. Last, we plan to investigate possibilities of “approximate provisioning”, where we lose some granularity of the possible assignments to variables but gain in performance.

Acknowledgements. This research has been partially funded by the Israeli Science Foundation, the Blavatnik Interdisciplinary Cyber Research Center (TAUICRC), the Blavatnik Computer Science Research Fund and Intel. The contribution of Yuval Moskovitch is part of Ph.D. thesis research conducted at Tel Aviv University.

REFERENCES

- [1] Amazon Inc. *Amazon Elastic Block Store (Amazon EBS)*. <https://aws.amazon.com/ebs/>, 2017.
- [2] Amazon Inc. *Amazon Elastic Compute Cloud (Amazon EC2)—Instance Types*. <https://aws.amazon.com/ec2/instance-types/>, 2017.
- [3] Y. Amsterdamer, S. B. Davidson, D. Deutch, T. Milo, J. Stoyanovich, and V. Tannen. Putting lipstick on pig: Enabling database-style workflow provenance. *PVLDB*, 5(4), 2011.
- [4] S. Assadi, S. Khanna, Y. Li, and V. Tannen. Algorithms for Provisioning Queries and Analytics. In *(ICDT 2016)*, volume 48, 2016.
- [5] C. Chen, H. T. Lehari, L. K. Loh, A. Alur, L. Jia, B. T. Loo, and W. Zhou. Distributed provenance compression. In *SIGMOD*, 2017.
- [6] D. Deutch, Z. G. Ives, T. Milo, and V. Tannen. Caravan: Provisioning for what-if analysis. In *CIDR*, 2013.
- [7] D. Deutch, Y. Moskovitch, and V. Tannen. Provenance-based analysis of data-centric processes. *VLDB J.*, 24(4), 2015.
- [8] A. Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4), 2001.
- [9] R. Ikeda, H. Park, and J. Widom. Provenance for generalized map and reduce workflows. In *CIDR*, 2011.
- [10] T. Imielinski and W. L. Jr. Incomplete information in relational databases. *J. ACM*, 31(4), 1984.
- [11] M. Interlandi, K. Shah, S. D. Tetali, M. A. Gulzar, S. Yoo, M. Kim, T. D. Millstein, and T. Condie. Titian: Data provenance support in spark. *PVLDB*, 9(3), 2015.
- [12] H. Karau, A. Konwinski, P. Wendell, and M. Zaharia. *Learning Spark: Lightning-Fast Big Data Analytics*. O’Reilly Media, Inc., 1st edition, 2015.
- [13] H. Park, R. Ikeda, and J. Widom. RAMP: A system for capturing and tracing provenance in mapreduce workflows. *PVLDB*, 4(12), 2011.
- [14] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.
- [15] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud’10*, 2010.

On Answering Why-Not Queries Against Scientific Workflow Provenance

Khalid Belhajjame

Université Paris-Dauphine, PSL Research University, CNRS, [UMR 7243], LAMSADE
Paris, France

Khalid.Belhajjame@dauphine.fr

ABSTRACT

Why-not queries help scientists understand why a given data item was not returned by the executions of a given workflow. While answering such queries has been investigated for relational databases, there is only one proposal in this area for workflow provenance, viz. the Why-Not algorithm. This algorithm makes the assumption that the modules implementing the steps of the workflow preserve the attributes of the input datasets. This is, however, not the case for all workflow modules. We drop this assumption, and show in this paper how the Web can be harvested to answer why-not queries against workflow provenance.

1 INTRODUCTION

Scientific workflows have been shown to facilitate and accelerate scientific data exploration and analysis in many areas of sciences [6]. A workflow can be viewed as an acyclic graph in which the nodes are modules that can be executed locally or remotely, and the edges specify the data dependencies between the constituent modules. Workflows have been utilized to model and enact in-silico experiments in a range of scientific fields, including proteomics, transcriptomics, metabolics, plant phenotyping, astronomy, and bio-medicine.

Fig. 1 illustrates an example of a simple workflow used for identifying the pathway associated with a given input metabolite (compound). Given a compound identifier, the first module returns a compound name, which is used to feed the second module to obtain the corresponding pathway.

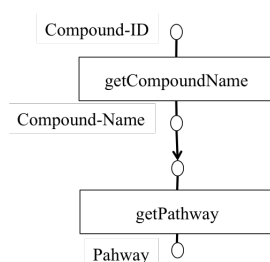


Figure 1: Example workflow.

Major workflow systems are instrumented to capture provenance information that records the data items used and generated by the workflow modules together with information specifying the lineage of such data items across the workflow execution.

Workflow provenance information can be utilized in a range of applications [8]. For example, it can be used to i)- estimate the quality of data based on the source data, ii)- determine the entity (author) to whom a given data item should be attributed, and

iii)- debug errors in the workflow execution. In this work, we focus on an application that has received little attention within the workflow provenance community, namely why-not queries. Given the provenance traces of workflow executions, the scientist may want to understand why a given result, e.g., his/her favorite protein, does not appear in the result of workflow executions.

1.1 Related Work

We distinguish between two classes of proposals for answering why-not queries: instance-based and module-based.

Instance-based. Proposals that fall into this category attempt to find the data items in the inputs that are responsible for the non appearance of a given data item in the result. More specifically, they determine the changes that need to be applied to the input datasets for the data item provided by the user to appear in the result. Artemis [9] and Missing-Answers [10] are examples of algorithms for processing instance-based why-not queries over relational databases. There is no proposal in the state of the art that investigates instance-based why-not queries for workflow provenance.

Module-based. Proposals in this class attempt to identify the modules that are responsible for the non-appearance of a given data item in the workflow results. The only proposal in this category for workflow provenance is the Why-Not algorithm proposed by Chapman and Jagadish [4]. Indeed, while why-not queries have been investigated in databases, where datasets are manipulated using (white boxes) query operators with known operational semantics, this is not the case for scientific workflows, where the steps of the workflow are implemented by black boxes, the behavior of which is not necessarily known. Using the Why-Not algorithm proposed by Chapman and Jagadish, the user query is expressed as a set of atomic predicates that are combined using AND and OR. An atomic predicate is evaluated over a single attribute of the Input datasets of the workflow. Chapman and Jagadish make the assumption that the attributes of the input datasets are preserved by the modules that compose the workflow. This is not the case, however, in the general case. For example, the modules in the workflow illustrated in Fig. 1 do not preserve the attribute of the input, viz. Compound – ID, in that the output of the first and the second module do not contain information about the compound identifier.

As well as identifying the reason why a result is missing, a number of proposals investigated changes that can be made to the query to include known missing results (see e.g., [1, 5]).

In this work, we focus on explaining why a data item is missing from the results of a data-driven workflow. In doing so, we drop the assumption made by Chapman and Jagadish, and propose a solution that can be utilized for answering why-not queries for workflow with modules that do not preserve attributes of the input datasets. Furthermore, unlike the Why-Not algorithm which is module-based, our proposal is hybrid in that it seeks

to answer instance- and module-based why-not queries. This makes our solution the only hybrid solution targeted for workflow provenance.¹

In the rest, we start by laying down the foundations in Sect. 2. We sketch our algorithm in Sect. 3. We then focus on operations that are central to our algorithm, namely determining if a module is picky, i.e., responsible for the non-appearance of an output data item, and determining the missing input data item for a given module in Sect. 4. Finally, we report on the results of a feasibility study that we conducted, and conclude the paper in Sect. 5.

2 FOUNDATIONS

We define a workflow WF as a directed acyclic graph in which the nodes correspond to modules, and the edges specify data flow dependencies. A data link connecting a module M_1 to a module M_2 specifies that the output produced by the invocation of the first is used as input to feed the execution of the latter.

The invocation of a module M , which we call module instance and denote by m , takes one or more data items as input and produce a data item as output. We write $m(d^{I_1}, \dots, d^{I_n}) = d^O$, where I_1, \dots, I_n represents the domain of values of the inputs of M and O the domain of values of the output. For the purpose of this work, we consider that the modules of a workflow are functions, in that they return the same output data item given the same input data items.

The execution of a workflow WF gives rise to a workflow instance wf, which takes one or more data items and produce as a result a data item. Major scientific workflow systems are instrumented to capture provenance information of workflow instances specifying the data items used and generated by the workflow, which can be utilized to track the lineage of the workflow results.

Typically, a scientist would execute a workflow WF multiple times, and then proceed to the exploration and analysis of the results. We are, therefore, interested in querying the datasets used and generated as a result of multiple executions (instances) of WF. We write $WF(D^{I_1}, \dots, D^{I_n}) = D_0$ to denote that collectively the instances of the workflow WF, took as input the datasets D^{I_1}, \dots, D^{I_n} and generated the dataset D_0 . Similarly, we write $M^{WF}(D^{I_1}, \dots, D^{I_n}) = D_0$ to denote that the invocations (instances) of the module M that took place with the instances of the workflow WF used the datasets D^{I_1}, \dots, D^{I_n} and generated the dataset D_0 .

Inverse of a Module. Central to our solution is the notion of the inverse of a module. The inverse of a module M , which we denote by M_{inv} takes as input data items that are type-compatible with the output of M , and delivers data items that are type-compatible with the inputs of M .

Picky module. A module M is picky with respect to a data item d if M_{inv} does not accept d as input. More specifically, M_{inv} throws an illegal input exception when its execution is fed d .

3 ANSWERING WHY-NOT QUERIES

3.1 Why-Not Queries

A user specifies a why-not query by specifying a data item, which has the same data type as the output of the last module of the workflow. Let such a module be $M^{WF}(D^{I_1}, \dots, D^{I_n}) = D_0$. Consider for the sake of simplicity that such a module output data items that are characterized by the attributes $\langle a_1, \dots, a_m \rangle$. A why-not query is a data item $\langle v_1, \dots, v_m \rangle$ that was not generated by M

¹There are existing hybrid solutions, but they are targeted for relational databases (see e.g., [2]).

when invoked within the workflow WF. That is, $\langle a_1, \dots, a_m \rangle \notin D_0$. We denote such a data item in what follows by $d_{\text{why-not}}$. Note that, in the general case, a workflow may have multiple final modules. Our solution is applicable to those workflows. To do so, our algorithm is applied to each output data item (why-not query) provided by the user.

3.2 Processing Why-Not Queries

To answer a why-not query, the modules of the workflow are explored from the sink to the source in a breadth-first fashion. To do so, we group the workflow modules into levels as illustrated in Figure 2. A module belongs to a level i if its outputs are connected to modules in levels $\leq i - 1$. Of course, this does not apply to the modules in level 0, the outputs of which carry the results of the workflow execution. The modules of a given level can be examined only if the examination of the modules of the previous level has completed.

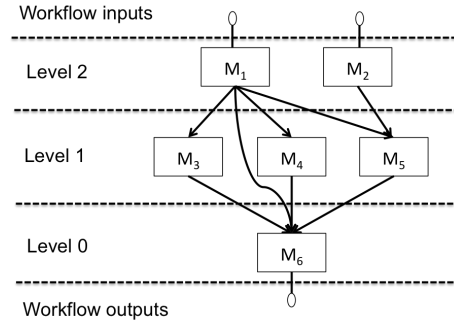


Figure 2: Workflow levels.

Algorithm 1, which we named Why-Not Detective, sketches the evaluation of a why-not query. It takes as input a data item $d_{\text{why-not}}$ specified by the user, and the workflow modules organized into levels starting from the sink WF_Modules. The modules of each level are examined to identify if the module is picky. Specifically, the inverse of the module in question M is examined to check if:

- It does not accept the corresponding data items that were generated by the inverse of the modules in the previous level. By corresponding data items, we mean data items generated by module parameters in the previous level, if any, that are connected to the output of the M module. The module M is flagged in this case as picky.
- It accepts the corresponding data items that were generated by the inverse of the modules in the previous modules. In this case, the data items the inverse of M produces are saved to be used to feed the inverse of the modules in the succeeding levels, if any. Notice here that the provenance of the workflow subjects to analysis will not contain such data items, otherwise, the workflow would have produced $d_{\text{why-not}}$.

The algorithm iterates over the modules of each level until i)- it finds modules of a given level that are picky, or ii)- it does not find any picky modules. In the first case, the algorithm stops (see line 15). Indeed, the modules of the succeeding level, or at least subset thereof, cannot be examined. This is because we will not have data items to probe the inverse of such modules with. The picky modules are therefore returned by the algorithm as the source of the non appearance of $d_{\text{why-not}}$ in the results. Note here that

our notion of picky module corresponds to the notion of *Frontier Picky Manipulation* in the Why-Not algorithm by Chapman and Jagadish [4]. In the second case (ii), the datasets used as input to the execution of the workflow are designated as the source of the non-appearance of $d_{\text{why-not}}$ in the results. Specifically, the data items returned by the inverses of the modules in the last level are designated as missing in the inputs datasets.

Example 3.1. Consider our example workflow in Fig. 1, and suppose that the scientists did not find the pathway identified by map00220 in the results of the provenance of the workflow. In this simple example, we have two levels composed of one module each. The algorithm starts by examining the getPathway module. The inverse of this module accepts the value map00220 and provides the compound name L – Argentine. This compound name is accepted by the inverse of getCompoundName, which in turn delivers the compound identifier cpd : C00062. The Why-Not detective concludes, in this case, that none of the modules is picky, and that the non appearance of the pathway map00220 in the results is due to the missing input data item cpd : C00062.

Algorithm 1 Why-Not Detective

Input: WF_Modules = $\{L_0, \dots, L_k\}$ // workflow modules grouped into levels (breadth) from the sink to the source.
 $d_{\text{why-not}}$ // missing workflow result.
Output: PickyModules // set of picky modules
MissingInputData // set of data items missing in the workflow input dataset

```

1: PickyModules =  $\emptyset$ 
2: MissingInputData =  $\emptyset$ 
3: Dcurrent =  $\{d_{\text{why-not}}\}$ 
4: Dcurrent+1 =  $\emptyset$ 
5: for Level in WF_Modules do
6:   for M in Level do
7:     if accept(inv(M), getInput(Dcurrent)) then
8:       datacurrent+1 += getResult(inv(M), getInput(Dcurrent))
9:     else
10:      PickyModules +=  $\{M\}$ 
11:    end if
12:    Dcurrent+1 += getResult(inv(M), getInput(Dcurrent))
13:  end for
14: if PickyModules  $\neq \emptyset$  then
15:   breakall
16: end if
17: Dcurrent = Dcurrent+1
18: end for

```

4 DETERMINING THE OUTPUT OF THE INVERSE MODULE

The central operations, which are repeatedly performed in the above algorithm, are the test of acceptance of data items by the inverse of a given module M , and the calculation of the results (data items) returned by the invocation of the inverse module.

To assess whether a module M_{inv} accepts a given data item d , we need to invoke M_{inv} using d . If the invocation of M_{inv} terminates successfully, then we can conclude that M_{inv} accepts d . Otherwise, if the execution of M_{inv} raises an illegal input exception, then we can conclude that M_{inv} does not accept d .

Unfortunately, we cannot perform this test, because we do not have access to M_{inv} . To address this issue, we harvest the (probably) biggest source of information, namely the Web. There are several proposals in the literature that attempted to extract

relational data from tables on the Web (see e.g., [3, 11]). Our objective is, however, different. We do not seek to transform all tabular HTML data into a structured format, but instead, identify the input data items that may be candidate for producing a given data item when used to feed a given module.

We adopt for this purpose a process, composed of three steps: i)- identifying candidate pages, ii)- extracting candidate input data items, and iii)- examining the candidate input data items.

4.1 Identifying Candidate Web Pages

To identify the web pages of interest, we make use of semantic annotations describing the input and output parameters of the module M . Indeed, a number of scientific modules are annotated with ontological concepts informing the semantic domain of the input and output parameters of the module, see e.g., bio-tools². We make use of semantic annotations, because often the names of the input and output parameters are non-informative and take values like in or out. Semantic annotations provide a crisper description of parameter types.

Consider for example that M has an input with a semantic domain c_{in} and an output of a semantic domain c_{out} . We issue a query with the following keywords: $\{c_{\text{in}}, c_{\text{out}}, d\}$ against a Web search engine. We are interested in finding the web pages W_c that contains all of the keywords. We call W_c candidate web pages. If the set W_c is empty, then M_{inv} is likely not to accept the data item d . Note that we say *likely*. This is partly because the Web, albeit a large data source, there is no guarantee of it being complete, and partly because the keywords used may fail to locate a Web page that contains the desired information, even if such a page exists.

4.2 Extracting Candidate Input Data Items

For each web page in W_c , we apply information extraction algorithms to retrieve data items that have c_{out} as a semantic domain. The problem we face here is that Web pages are unstructured. Solutions that have been proposed for information extraction, e.g., [3, 7, 11] can be used for this purpose.

As well as the above proposals, in the context of our work, we use recognizers. Indeed, many of the semantic domains, e.g., pathways, enzymes, proteins, etc., in the scientific fields are associated with recognizers, able to identify the semantic domain of a given raw text/string. This applies particularly to accessions, which can be seen as scientific identifiers for entities such proteins, RNAs, etc., as well as other more complex structures such as sequence entries, e.g., Fasta format, IPR entry, etc. Therefore, if c_{out} is associated with a recognizer, then we apply the recognizer to each of the web pages in W_c . This will result in a list of candidate data items D_{in}^c .

4.3 Examining Candidate Input Data Items

We use the data items in D_{in}^c to feed the execution of the module M . If an execution that takes an input data item d_{in} yields a successful execution of M and produces a result d , then we can conclude that the inverse M_{inv} accepts d and it delivers as a result d_{in} . Note that in the general case, multiple input data items can be associated with d . For the purpose of this work, however, we assume that the module M and its inverse are functions.

If, on the other hand, none of the candidates in D_{in}^c yields d when invoking M , then we conclude that M_{inv} is likely not to accept d .

²<https://bio.tools>

5 FEASIBILITY STUDY

The approach we have just described raises the following question. *Is the algorithm proposed able to identify the reason why a given data item does not appear in the workflow results?* More specifically, *How effective is our solution in identifying picky modules and missing input data items?*

To answer the above questions, we run a feasibility experiment, in which we used a sample of 6 real-world workflows from the myExperiment repository³. We selected workflows that involve deterministic modules, which mean modules that deliver the same result (if any) given the same input. We did not consider workflows that include modules performing data mining operations, for instance. We have also selected workflows for which the inverse modules are also deterministic functions.

5.1 Set-up

We have executed each workflow using example data inputs provided by the workflow authors. In doing so, we used the Taverna workflow systems [12]. We then specified two kinds of queries for each workflow:

- Instance-based why-not query. To assess the ability of the algorithm in answering this type of queries, we randomly selected an output data item d that was returned by the workflow executions. Next, we used our algorithm to see if it is able to reconstruct the lineage of d by harvesting the web to identify the input data items that were responsible for its derivation. We then compared the lineage reconstructed for d with the lineage available in the provenance of the workflow executions previously recorded by the workflow system. This allows us to see if our algorithm was able to successfully identify the input data items responsible for the derivation of d .
- Module-based why-not query This kind of query is used to assess if the algorithm is able to identify picky modules. That is modules that are responsible for the non deliverance of a given data item d by the workflow executions. To do so, for each workflow, we identified a data item that we know it cannot be delivered by the workflow because of a given (picky) module. We then used our algorithm to assess if it is able to identify such a module.

In total we had 6 queries of the first kind, which we denote by $\{q_1^+, \dots, q_6^+\}$, and 6 queries of the second kind, which we denote by $\{q_1^-, \dots, q_6^-\}$. The + sign indicates that we should be able to reconstruct the provenance of the why-not query up to the workflow input, and the - sign indicates that we should not be able to do so, and instead identify the picky module.

5.2 Results

Of the queries $\{q_1^+, \dots, q_6^+\}$, our algorithm was able to successfully constructs the provenance of the why-not query up to the workflow input for 3 queries. Most of the modules composing these workflows, namely 8 out of 11, provides information about the input and output datasets on the Web using Tabular formats.

After examination of the three remaining workflows, we found that one them utilizes proprietary data sources, the content of which is not accessible on the surface web. The last two workflows, on the other hand, contain modules that manipulate excerpt from HTML web pages. Because of this, our algorithm was not able to find the content on the Web of the input and output

of those modules. However, we made a key observation thanks to these two workflows that will allow us to improve the quality of the results delivered by our algorithm. Indeed, these modules that were problematic for our algorithm perform format transformation, which we refer to as Shims in scientific workflows. Such modules could be ignored (skipped) by our algorithm. For example, once ignored, our algorithm was able to identify the inputs data items of the remaining modules in the workflows.

We also measured the number of Top-k web pages that needed to be examined to identify the input data item corresponding to a given output data item. On average, we needed to examine the content of the 4 top web pages returned by the key-word search engine⁴. In several cases, however, the top web page was the right one, in the sense that it contained the input data item we are after.

Regarding the queries $\{q_1^-, \dots, q_6^-\}$, our algorithm was more successful in the sense that it was able to correctly identify 4 picky modules out of 6. For two remaining workflows, the module that was identified as picky by our algorithm was not the correct one. After examination, it transpired that for certain modules the corresponding data item could not be found on the web. Again this issue was due to shims modules the input and output data items are not published on the Web.

To sum up, this small feasibility study has shown that our method is promising. It has also brought some insights into the way our solution can be improved. Our ongoing work includes: i)- tuning our algorithm to deal with shims modules in a workflow, ii)- explore new source of information for identifying picky modules, iii)- extending our solution to cases where the inverse of a module is not a function, and iv)- an experiment involving a large number of scientific workflows.

REFERENCES

- [1] S. S. Bhowmick, A. Sun, and B. Q. Truong. 2013. Why not, WINE?: towards answering why-not questions in social image search. In *ACM Multimedia Conference, MM '13, Barcelona, Spain, October 21-25, 2013*. ACM, 917–926. <https://doi.org/10.1145/2502081.2502098>
- [2] Nicole Bidoit, Melanie Herschel, and Katerina Tzompanaki. 2014. Query-Based Why-Not Provenance with NedExplains. In *Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014*. OpenProceedings.org, 145–156.
- [3] Michael J. Cafarella, Alon Y. Halevy, Daisy Zhe Wang, Eugene Wu, and Yang Zhang. 2008. WebTables: exploring the power of tables on the web. *PVLDB* 1, 1 (2008), 538–549. <http://www.vldb.org/pvldb/1/1453916.pdf>
- [4] Adriane Chapman and H. V. Jagadish. 2009. Why not?. In *Proceedings of SIGMOD*. ACM, 523–534.
- [5] L. Chen, J. Xu, X. Lin, C. S. Jensen, and H. Hu. 2016. Answering why-not spatial keyword top-k queries via keyword adaptation. In *ICDE. IEEE-CS*, 697–708. <https://doi.org/10.1109/ICDE.2016.7498282>
- [6] Rafael Ferreira da Silva, Ewa Deelman, Rosa Filgueira, et al. 2016. Automating environmental computing applications with scientific workflows. In *Proceedings of eScience*. IEEE, 400–406.
- [7] Maeda F. Hanafi, Azza Abouzied, Laura Chiticariu, and Yunyao Li. 2017. SEER: Auto-Generating Information Extraction Rules from User-Specified Examples. In *Proceedings of the CHI Conference*. ACM, 6672–6682.
- [8] Melanie Herschel, Ralf Diestelkämper, and Houssein Ben Lahmar. 2017. A survey on provenance: What for? What form? What from? *VLDB J.* 26, 6 (2017), 881–906.
- [9] Melanie Herschel and Mauricio A. Hernández. 2010. Explaining Missing Answers to SPJUA Queries. *PVLDB* 3, 1 (2010), 185–196.
- [10] Jiansheng Huang, Ting Chen, AnHai Doan, and Jeffrey F. Naughton. 2008. On the provenance of non-answers to queries over extracted data. *PVLDB* 1, 1 (2008), 736–747.
- [11] Rakesh Pimplikar and Sunita Sarawagi. 2012. Answering Table Queries on the Web using Column Keywords. *PVLDB* 5, 10 (2012), 908–919.
- [12] K. Wolstencroft, R. Haines, D. Fellows, et al. 2013. The Taverna workflow suite: designing and executing workflows of Web Services on the desktop, web or in the cloud. *Nucleic acids research* (2013), W557–W561.

³www.myexperiment.org

⁴We used the Google search engine for our experiment.

PRoST: Distributed Execution of SPARQL Queries Using Mixed Partitioning Strategies

Matteo Cossu
 Institute for Computer Science
 University of Freiburg
 elcossu@gmail.com

Michael Färber
 Institute for Computer Science
 University of Freiburg
 michael.farber@cs.uni-freiburg.de

Georg Lausen
 Institute for Computer Science
 University of Freiburg
 lausen@informatik.uni-freiburg.de

ABSTRACT

The rapidly growing size of RDF graphs in recent years necessitates distributed storage and parallel processing strategies. To obtain efficient query processing using computer clusters a wide variety of different approaches have been proposed. Related to the approach presented in the current paper are systems built on top of Hadoop HDFS, for example using Apache Accumulo or using Apache Spark. We present a new RDF store called PRoST (Partitioned RDF on Spark Tables) based on Apache Spark. PRoST introduces an innovative strategy that combines the Vertical Partitioning approach with the Property Table, two preexisting models for storing RDF datasets. We demonstrate that our proposal outperforms state-of-the-art systems w.r.t. the runtime for a wide range of query types and without any extensive precomputing phase.

1 INTRODUCTION

Organizations and institutions increasingly see a need to represent their data in a semantically-structured way [7] and, thus, rely on the RDF data model. As the data represented in RDF constantly grows in size, storing and querying these very large RDF graphs becomes a major challenge. In order to increase the query efficiency, many different approaches have been proposed e.g. [1, 10, 13, 15]. Most of the current solutions use DBMS-based systems and map SPARQL queries to SQL queries for the retrieval. Besides the systems which are implemented for single-node machines, such as RDF-3X [11], Sesame [6] and Jena [16], systems designed for distributed environments are increasingly being used. They allow to scale up the storage and to provide parallel query execution capabilities, which is essential to handle very large data. Systems of distributed environments use in particular Hadoop technologies: S2RDF [15] and SPARQLGX [8] are implemented on top of Apache Spark, Rya [13] on top of Apache Accumulo, and Sempala [14] on top of Impala.

Typically, these systems are optimized for specific query patterns and their data loading times are often sacrificed for better querying performance. Thus, there is need for a distributed RDF store with better performance on a wide range of query types, without renouncing a rapid loading phase.

In this work, we describe the approach of PRoST¹ (Partitioned RDF on Spark Tables), that aims to improve the efficiency of queries on RDF data, thereby covering a wide range of query types. Instead of building a standalone system, such as [12], AdPart [2] or TriAD [9], PRoST is based on reliable Hadoop technologies for storing and processing the data and therefore its

¹The source code of PRoST is online available at <https://github.com/tf-dbis-uni-freiburg/PRoST>.

performance depends more on the general approach than on the details of the implementation (e.g. memory management, network protocols). In this paper, we focus on querying RDF efficiently by means of Apache Spark², as it emerged as an important component of the Hadoop ecosystem: It offers general purpose data processing functions that exploit efficiently the main memory – differently from MapReduce that relies mainly on disk operations. This characteristic makes Spark very fast in practice and able to compute complex queries on large RDF graphs. In particular, it is possible to obtain high querying performances using Spark SQL, a module of Spark that adds well-established DBMS techniques to the distributed computation framework.

Our main contribution is proposing a new storage model for loading RDF graphs into Hadoop, including an appropriate strategy for translating SPARQL queries into Spark execution plans. We show that combining the so far separately applied strategies for representing RDF data in Hadoop – Vertical Partitioning [1] and the strategy of using Property Tables [16] – can be implemented with relatively little effort and leads to superior querying performances in many cases, as our evaluation results show.

The paper is structured as follows: In Section 2, we present an overview of similar systems for querying RDF. In Section 3, we outline our approach for modeling the data, our strategy for translating SPARQL queries, and the techniques for improving the performances of PRoST. We summarize our evaluation results in Section 4 and, in Section 5, we conclude and present ideas for future implementations of PRoST.

2 RELATED WORK

Due to space limitations, in the following we restrict ourselves to Hadoop-based RDF querying approaches, as they are the most similar to our system PRoST. S2RDF [15] is a "SPARQL processor" based on Spark SQL. It introduces a data partitioning approach that extends Vertical Partitioning [1] with additional tables, containing precomputed semi-joins. As a consequence, many intermediate results of queries are already computed and can be used to shorten the retrieval time. However, S2RDF trades off the performances with disk space and loading time. For datasets with a large number of properties (e.g., DBpedia [5]), the time required may make the loading unfeasible.

SPARQLGX [8] is another system for distributed SPARQL queries that uses Vertical Partitioning. SPARQLGX compiles the queries directly into Spark operations. The system relies entirely on its own statistics to optimize the computation, in particular for the order of the joins. Differently from S2RDF and PRoST, SPARQLGX does not use Spark SQL.

Rya [13] is a popular RDF management system, developed by an active open-source community and currently an incubating project at Apache. It is built on top of Apache Accumulo³, a distributed key-value datastore for Hadoop. Since Accumulo

²<https://spark.apache.org>.

³<https://accumulo.apache.org/>.

keeps all its information sorted and indexed by key, Rya stores whole RDF triples as keys. For this reason, single triples or short ranges can be accessed very fast. However, considering that the triple table model of Rya contains three columns, it requires to replicate the data multiple times in order to exploit the indexes over all the possible elements. Rya uses some query optimization, e.g. joins reordering, but it lacks of the powerful in-memory data processing that make, in practice, other systems faster.

3 APPROACH

The main idea behind PRoST is to store the data twice, partitioned in two different ways. Each one of the two data structures can be more beneficial (in terms of performance) than the other regarding certain query types, mainly because of the different location of the data in the cluster. Our system tries to exploit this characteristic. We do not limit ourselves to choose the single best model out of these two for each query, but we split a query in several parts, such that each sub-query can be executed with the most appropriate approach. At the end, all the parts can be joined together to produce the final result. In this way, our approach achieves better performance than when running only on one of the two representations, as it leverages the synergy effects of both models.

3.1 PRoST Data Model

The two data structures we use to store an RDF graph are *Vertical Partitioning* and *Property Table*. *Vertical Partitioning* (VP) is an approach proposed by Abadi et al. [1]. It has a strong popularity among RDF storage systems and it is the main choice of the state-of-the-art approaches S2RDF [15] and SPARQLGX [8]. The main concept is to create a table for each distinct predicate of the input graph, containing all the tuples (subject, object) that are connected by that predicate. The vertical partitioning approach is at the same time powerful and simple, but it has the drawback that it needs a number of joins proportional to the number of triple patterns in the query (precisely, the number of triple patterns minus one). Even with this disadvantage, because the VP tables are narrow and often small, VP is a valid and practical choice.

The *Property Table* (PT) is a data scheme that was introduced in the implementation of Jena2 [16], an influential toolkit for RDF graph processing, and Sempala [14], another SPARQL processor built on top of the Impala framework. A PT consists of a unique table where each row contains a distinct subject and all the object values for that subject, stored in columns identified by the property to which they belong.

The most important advantage of PT, when compared with VP, is that several joins can be avoided when some of the triple patterns in a query share the same subject. In this case, part of the query can be executed by a simpler select operator. Therefore, this approach is beneficial for queries where all the triple patterns have the same subject variable, usually called *star* queries. However, as observed by Abadi et al. [1], the PT has some issues, in particular the number of NULLs and the multi-valued attributes. Since not every possible pair subject-predicate has a corresponding object value, the PT potentially contains a very large number of NULLs. We solve this problem by storing the table in *Parquet*⁴, a format that uses run-length encoding. The other problem of the PT is the presence of multi-valued properties, i.e., when more than one different object value exists for at least one subject. In this case, the values are stored using lists

⁴<http://parquet.apache.org>

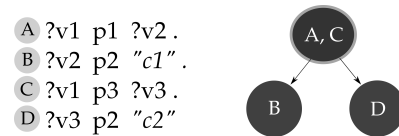


Figure 1: Example of a conjunction of triple patterns translated to a Join Tree. The root node uses the Property Table, the others Vertical Partitioning.

that need to be flattened when executing operations on these attributes. This particular situation produces an overhead in respect to the VP model, significant if we use the PT for a single attribute, but negligible if compared to the benefit of avoiding more than one join to compute the same query.

We have several advantages (e.g. efficient compression) in using the columnar format *Parquet* to store the PT, but we then introduce also the undesirable possibility that the data belonging to a single subject could be split into different nodes. In order to keep benefiting from the structure of the property table, we partition horizontally on the subject column. In this way, we ensure that every row is stored entirely in the same node.

3.2 Query Strategies

Having two data structures at the same time requires an abstraction to represent the queries. For this reason, we define an intermediate format that we call *Join Tree*. Each node of this tree represents a sub-query extracted from a VP table or from the PT. Using Spark, PRoST can execute a query by calculating the intermediate results from the nodes and then joining them together in a bottom-up fashion.

We now explain how we translate SPARQL queries into the *Join Tree* format. For the sake of clarity, we consider queries with a unique *basic graph pattern* without filter, which are a conjunction of triple patterns. The triple patterns with the same subject are grouped together and translated into a single node with a special label, denoting that we should use the property table in this case. All the other groups with a single triple pattern are translated to nodes that will use the vertical partitioning tables. An example is shown in Figure 1. There could be a join edge between every pair of nodes that shares a variable, it means that more than one *Join Tree* translation is possible for a single query. Since the tree structure influences the final performances, choosing carefully the *Join Tree* is important for the quality of the system. For this reason, we choose a tree guided by simple statistics, as explained in the following section.

3.3 Statistics-based Optimization

In the context of our work, joins are the most expensive operators because they need large portion of the data to be shuffled across the network. The order of the joins is important to limit this issue, and therefore to speed up the calculation. An effective way is sorting the joins using statistics of the input graph. The *Join Tree* of a query decides indirectly the order in which the operations are computed. For example, the leaves are computed first, and the root node is the final one. The statistics we use, simple but effective in practice, are (1) the total number of triples and (2) the number of distinct subjects for each predicate. They are calculated during the loading phase without any significant overhead.

As a general rule of thumb, we want to compute the joins involving few data or with a high selectivity first (high priority), and finally the ones with many tuples (low priority). The priority value of a node of the Join Tree is decided by the following criteria:

- Triple patterns containing literals are scored with the highest priority. The presence of a literal is a strong constraint that limit the number of resulting tuples. Therefore, it is a good approach to push down these kind of nodes.
- A triple pattern of which the underlying data contains many tuples will be scored proportionally. For instance, a triple with the highest number of tuples will have also the lowest priority and it will be the root of the tree. This number is also adjusted according the number of distinct subjects for that predicate.
- The priority of a node containing data belonging to the Property Table is scored taking in account all its triple patterns. However, the presence of a triple pattern with a literal is weighted heavily.

In addition to our effort, Spark SQL’s Catalyst Optimizer [4] uses its internal heuristics to improve query performances further. The trees are not substantially changed, but Spark intervenes in producing optimized physical plans, since it knows the concrete location of the data on the cluster. In particular, the optimizer can choose the type of joins to perform, for example if one of the relations involved is small, a broadcast join will be performed.

4 EVALUATION RESULTS

4.1 Benchmark Environment

We perform our tests on a small cluster of 10 machines connected via Gigabit Ethernet connection. Each machine is equipped with 32GB of memory, 4TB of disk space and with a 6 Core Intel Xeon E5-2420 processor. The cluster runs Cloudera CDH 5.11.0 on Ubuntu 14.04 and Spark 2.1.0. Since one machine is the master, Spark uses only 9 workers. The executor’s memory is 21GB. We use the WatDiv [3] dataset provided by the Waterloo SPARQL Diversity Test Suite 7. It is developed in order to measure how an RDF data management system performs across a wide variety of SPARQL queries with varying characteristics and has already been applied in the evaluations of the comparable systems. Given WatDiv we generate a dataset containing around 100 Million RDF triples (5.16 GB) and we evaluate PRoST with the given *WatDiv basic query set*. It contains queries of varying shape and selectivity in order to model different scenarios. The queries are grouped into the following subsets:

- **C**: Complex shaped queries.
- **F (F1, F2, F3, F4, F5)**: Snowflake shaped queries.
- **L (L1, L2, L3, L4, L5)**: Linear shaped queries.
- **S (S1, S2, S3, S4, S5, S6, S7)**: Star shaped queries.

4.2 Loading

As we can see from Table 1, the storage size used by PRoST decreases when being stored in HDFS. This is a direct consequence of the compression used by the *Parquet* format.

The database size of S2RDF is the largest in this analysis, since its model not only needs the creation of VP tables, but also many additional ones containing the results of its precomputations. SPARQLGX occupies minimal space because it uses only Vertical Partitioning, where instead PRoST requires double SPARQLGX’s size for having in addition the Property Table. The loading time of

System	Size	Time
PRoST	2.1 GB	25m 32s
SPARQLGX	0.9 GB	20m 01s
S2RDF	6.2 GB	3h 11m 44s
Rya	3.1 GB	41m 32s

Table 1: Size and loading times using WatDiv100M

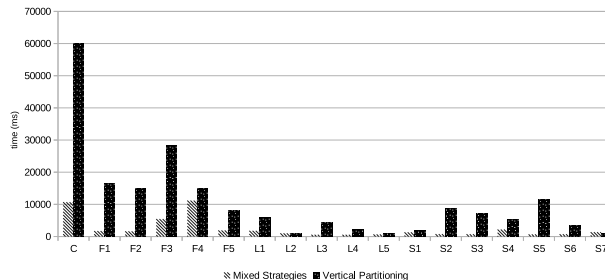


Figure 2: Querying time for WatDiv100M with only Vertical Partitioning and with the mixed strategy.

PRoST is similar to SPARQLGX and around an order of magnitude shorter than S2RDF. Similarly to what we said before for the database size, S2RDF’s model requires longer times because of its extensive precomputations.

4.3 Vertical Partitioning vs Mixed Strategy

One of our central points is the introduction of the Property Table alongside Vertical Partitioning. As a consequence, we evaluated the impact of this addition on query performances. In Figure 2, we show the times to compare the query set from WatDiv100M with Vertical Partitioning only and with the additional support of the Property Table.

The chart shows clearly that the introduction of the Property Table has a strong positive impact on performances. For almost every type of query this version outperforms abundantly the simple Vertical Partitioning approach. The new strategy is effective for Star queries (S), where most of the triples share the same subject, as well as Complex (C) and Snowflake (F) queries, that have more than one different subject. For some of the Linear queries (L), the results are very similar between the two versions. This result comes from the fact that Linear queries contain mostly triples with distinct subject variables, translated using mostly Vertical Partitioning.

4.4 PRoST vs Other Systems

To better evaluate our system we compared it in the same environment with other similar solutions: S2RDF, Rya and SPARQLGX. In Figure 3 we show the querying times on the WatDiv100M dataset with the other systems and our implementation. Note that we used a logarithmic scale, since the large differences would prevent us from visualizing these results well. Compared to S2RDF, PRoST is faster for the queries F2, S1, S3 and S5 but slower for the remaining queries. In particular, for the complex queries (C) and some of the snowflake ones (F3, F4) the S2RDF’s approach makes it faster by a considerable margin. These differences can be explained with the extensive precomputations of S2RDF, that heavily decrease the processing time for joins between VP tables. However we have to keep in mind that S2RDF can reach

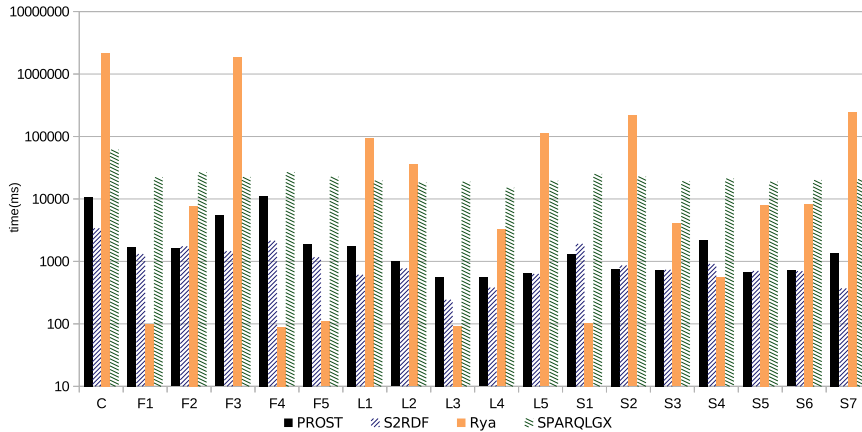


Figure 3: Querying time for WatDiv100M with PROST, S2RDF, Rya and SPARQLGX. The scale is logarithmic.

Queries	ProST	S2RDF	Rya	SPARQLGX
Complex	9,364ms	3,392ms	2,195,322ms	61,363ms
Snowflake	5,923ms	1,564ms	369,016ms	24,046ms
Linear	2,419ms	527ms	49,044ms	18,254ms
Star	1,195ms	884ms	6,9606ms	2,1046ms

Table 2: Average querying time grouped by type of query.

these results because of a possibly large loading time, that, for some datasets, may make it less attractive in practice. In contrast, PROST relies on a faster loading phase and its performances does not depend on the particular input graph, i.e. number of predicates.

For some queries, Rya is very fast and outperforms PROST but in other cases it suffers from significant slowdowns, that are several orders of magnitude slower. Therefore, its average query time is the worst of the systems considered, as shown in Table 2. The queries in which Rya performs very well have in common that their computation involves only few intermediate results. As a matter of fact, Rya uses powerful indexing methods but it lacks of more sophisticated approaches to calculate joins. For SPARQLGX, almost all the queries are computed in around twenty seconds, except the Complex ones (C) that have a higher number of resulting tuples. PROST outperforms SPARQLGX in every case, mostly by around an order of magnitude. Since SPARQLGX uses simple vertical partitioning, this difference confirms further the improvements produced by the introduction of the new data organization approach and also increases the validity of the implementation choices of PROST. The evaluation results presented in [8] are obtained using a virtual cluster of 10 nodes on 2 physical machines. In particular, when SPARQLGX is compared to S2RDF, their results are not consistent with the results from our experiment, obtained on a cluster of 10 physical machines.

5 CONCLUSION AND FUTURE WORK

In this paper, we presented PROST, a distributed system for RDF storage and SPARQL querying built on top of Apache Spark. With PROST we introduce an innovative data structure for partitioning RDF data that, to the best of our knowledge, for the first time combines efficiently two pre-existing approaches, namely Vertical Partitioning and the Property Table. Our evaluations, in which we compare PROST on a Hadoop cluster with some state-of-the-art systems Rya, SPARQLGX, and S2RDF, show that PROST

outperforms Rya and SPARQLGX in terms of querying time to a large extent and that it achieves similar results to S2RDF. Notably, PROST shows consistently good results for every type of query and is not limited to datasets with certain characteristics to keep the pre-computation feasible. Therefore, our approach is in particular appropriate for real-world applications, for which the query type and the dataset are unknown a priori.

For future work, we considered further improvements in terms of both storage and querying performance. For example, a promising step might be to add another Property Table where, instead of the subjects, the rows would be created around objects. This could be beneficial for triple patterns that share the same object. Another possible improvement would be to collect more precise statistics of the input dataset in order to produce better trees and, hence, a less expensive retrieval.

REFERENCES

- [1] Daniel J Abadi et al. 2007. Scalable semantic web data management using vertical partitioning. In *Proceedings of VLDB*. VLDB Endowment, 411–422.
- [2] Ibrahim Abdelaziz et al. 2017. Combining Vertex-centric Graph Processing with SPARQL for Large-scale RDF Data Analytics. *IEEE Transactions on Parallel and Distributed Systems* (2017).
- [3] Güneş Aluç et al. 2014. Diversified stress testing of RDF data management systems. In *Proceedings of ISWC*. Springer, 197–212.
- [4] Michael Armbrust et al. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of ACM SIGMOD*. ACM, 1383–1394.
- [5] Sören Auer et al. 2007. Dbpedia: A nucleus for a web of open data. *The semantic web* (2007), 722–735.
- [6] Jeen Broekstra et al. 2002. Sesame: A generic architecture for storing and querying rdf and rdf schema. In *Proceedings of ISWC*. Springer, 54–68.
- [7] Michael Färber et al. 2017 (preprint). Linked Data Quality of DBpedia, Freebase, OpenCyc, Wikidata, and YAGO. *Semantic Web Journal* (2017 (preprint)).
- [8] Damien Graux et al. 2016. Sparqlgx: Efficient distributed evaluation of sparql with apache spark. In *Proceedings of ISWC*. Springer, 80–87.
- [9] Sairam Gurajada et al. 2014. TriAD: a distributed shared-nothing RDF engine based on asynchronous message passing. In *Proceedings of ACM SIGMOD*. ACM, 289–300.
- [10] Kisung Lee and Ling Liu. 2013. Scaling queries over big RDF graphs with semantic hash partitioning. *Proceedings of VLDB Endowment* 6, 14 (2013), 1894–1905.
- [11] Thomas Neumann and Gerhard Weikum. 2008. RDF-3X: a RISC-style engine for RDF. *Proceedings of VLDB Endowment* 1, 1 (2008), 647–659.
- [12] Anthony Potter et al. 2016. Distributed RDF Query Answering with Dynamic Data Exchange. In *Proceedings of ISWC*. 480–497.
- [13] Roshan Punnoose et al. 2012. Rya: a scalable RDF triple store for the clouds. In *Proceedings of 1st IWCI*. ACM, 4.
- [14] Alexander Schätzle et al. 2014. Sempala: interactive SPARQL query processing on hadoop. In *Proceedings of ISWC*. Springer, 164–179.
- [15] Alexander Schätzle et al. 2016. S2RDF: RDF querying with SPARQL on spark. *Proceedings of VLDB Endowment* 9, 10 (2016), 804–815.
- [16] Kevin Wilkinson. 2006. Jena Property Table Implementation. In *Proceedings of SSWKBs*.

Deep Integration of Machine Learning Into Column Stores

Mark Raasveldt
CWI
Amsterdam
m.raasveldt@cwi.nl

Hannes Mühleisen
CWI
Amsterdam
hannes@cwi.nl

Pedro Holanda
CWI
Amsterdam
holanda@cwi.nl

Stefan Manegold
CWI
Amsterdam
manegold@cwi.nl

ABSTRACT

We leverage vectorized User-Defined Functions (UDFs) to efficiently integrate unchanged machine learning pipelines into an analytical data management system. The entire pipelines including data, models, parameters and evaluation outcomes are stored and executed inside the database system. Experiments using our MonetDB/Python UDFs show greatly improved performance due to reduced data movement and parallel processing opportunities. In addition, this integration enables meta-analysis of models using relational queries.

1 INTRODUCTION

The ad-hoc nature of data analysis pipelines using complex statistical or machine learning algorithms bundled with large volumes of data quickly leads to a latent need for data management systems [9]: 1) Manually managing large datasets as flat files quickly becomes cumbersome and error-prone, and additionally introduces many difficulties when multiple people are working with the same data or when the data has to be written to or updated. 2) Loading data from structured text files (e.g. XML or CSV) is very inefficient, as the data has to be parsed and converted to native binary formats before being subjected to analysis. Files are read from disk every single time an analysis pipeline is run, significantly increasing the time necessary to run these pipelines. These problems are exasperated as analysis pipelines are evaluated and moved towards production use.

These problems can be solved by using existing relational database management technologies. However, it has traditionally been very difficult to combine analytical tools with relational databases. The standard approach of running a separate database server and connecting with it through a socket connection is very inefficient and introduces severe bottlenecks when working with large amounts of data [15]. On the other end of the spectrum, in-database processing techniques have typically been cumbersome and difficult to use. Rewriting analytical pipelines in plain SQL is non-trivial and the subject of research papers [10]. Traditional scalar user-defined functions (UDFs) as supported by “mainstream” relational data management systems are difficult to utilize for complex machine learning tasks where a view on the entire dataset is required.

Recently, vectorized UDFs [8, 14] have been proposed that allow for efficient and flexible integration of popular analytical tools inside column-store databases. By utilizing these UDFs, existing complex analytical pipelines can be moved inside the

database. This allows us to gain all the advantages of storing data inside a relational database, while still having flexible and easy-to-use analytical tools available.

An additional benefit of training and using machine learning models directly in the database is that it is possible to persist both models and metadata (e.g. classification scores on test sets) in the database. Standard relational queries can then be used to apply the trained models to data. This allows for example to compare and combine output from multiple models, each specialized for certain classification tasks. Also, it is possible to classify the same data using multiple models and use the result of the model that reports the highest confidence.

In this paper we showcase how we can use vectorized user-defined functions to efficiently integrate a complex analysis pipeline inside a columnar database management system. We show how we can train models directly inside the database, and how to store the models and subsequently use them to classify data without having to export the data from the database system.

The main contributions of this paper are:

- We show how traditional classification models can be integrated into a column-store relational database management system.
- We describe how models can be stored inside the database system and how these models can then be used to efficiently and flexibly classify data.
- We experimentally show the performance benefit of directly running the models inside the database system versus loading the data from structured text, binary files or using database client protocols.

Outline. This paper is organized as follows: Section 2 discusses related work. Section 3 presents our integration approach, followed by a concrete use-case and performance results in Section 4. In Section 5 we draw our conclusions and discuss future work.

2 RELATED WORK

There is a variety of related work on combining relational database systems with machine learning pipelines. In this section we will present the most recent related work regarding the integration of machine learning through UDFs and model management systems and compare them with our solution.

2.1 Machine Learning Integration

Integrating existing Database Management Systems and machine learning algorithms has been a long standing problem due to the complexity of implementing the machine learning code inside a DBMS.

Early work [2, 16] on this focuses on rewriting analytical algorithms into portable SQL code. This allows the pipelines to be executed within any database system without requiring database-specific modifications. However, rewriting complex analytical pipelines in SQL requires a lot of manual effort and might not be possible for certain algorithms because SQL is not a Turing complete language.

In Ordonez et al. [12], machine learning algorithms are translated to either C, C++ or C# code (depending on the DBMS language support) and inserted into UDFs. As a consequence they achieve high performance when analyzing large data sets compared to external data analysis tools, as data movement is mitigated. However, these algorithm must be coded in one of the previously listed languages. This often results in the need for rewriting code, because most prominent machine learning libraries are usually available in scripting languages (e.g., Python and R). In our solution we allow the developer to use popular scripting languages *together with their entire ecosystem of data analytics packages* as UDFs in MonetDB.

Other work [4, 6, 7] focuses on more templated approaches for machine learning integration to reduce the necessity of code rewriting. However, the main disadvantage of these methods is that they only work for a limited subset of algorithms, which limits their applicability to general machine learning tasks.

2.2 Machine Learning Model Management

When training and using a variety of models the problem of managing these models arises. This problem is exasperated because most Machine Learning Systems do not provide support for storing and querying their models. Due to these issues, data scientists quickly lose track of their models.

In Vartak et al. [19], a system called ModelDB is introduced that can be used for storing, tracking and managing machine learning models in their native environment. This allows data scientists to use SQL to query their models based on their meta-data (e.g., hyperparameters, parameters) and quality metrics (e.g., accuracy). It also has the option to store the used train/test data sets for each model. However, since ModelDB only stores the models in their native environment, it does not provide a solution for coupling machine learning applications with traditional relational databases.

3 MACHINE LEARNING INTEGRATION

Machine learning pipelines consist of three stages [5].

- (1) **Preprocessing.** In this stage, the raw data is loaded and cleaned. The data is normalized, and any inconsistencies from incorrect or missing measurements are corrected for or removed.
- (2) **Training and Verification.** In this stage, the cleaned data is used to train the model. Typically the training set is divided into parts, and techniques like cross validation are used to prevent overfitting the model.
- (3) **Classification.** In the final stage, the trained model is used to classify new data. In this stage, the model can still be refined further based on new data or new properties of the data.

The preprocessing stage can often be performed entirely within traditional database management systems. Loading data and simple cleaning operations such as missing value removal can be done using standard SQL queries. However, when more advanced

preprocessing such as interpolation is required, user-defined functions can be used to simplify this step.

The real challenge of integrating these pipelines into databases, however, is implementing the machine-learning models. The models rely on complex math operations and iterative refinement, which are not supported by standards-complaint SQL.

There are many libraries and packages in vectorized scripting languages that implement common machine learning and classification models, such as TensorFlow [1] and Sci-Kit Learn [13]. Using vectorized user-defined functions, we can plug these libraries into the database. However, the typical processing pipelines must be adjusted so they can fit into a SQL workflow. In this section, we will describe how these analytical pipelines can be integrated into traditional database management systems through the use of user-defined functions.

3.1 Training

To train a classification model, we take a set of annotated data as input and use the annotations to find patterns in the data. After learning these patterns, the trained model can accurately classify un-annotated data.

The training pipeline therefore takes as input a set of columns representing the data, and a single column representing the classes of the data. This will be the input to our user-defined function. The output of this stage of the pipeline is the trained model, which will be the output of our UDF. The actual creation and training of the model will happen inside the function.

Model Storage. Models exist as in-memory objects within the scripting language. However, they can be serialized to a binary format for persistent storage on disk. In Python, this is done using the pickle library. In order to store the objects in the database we need to serialize the objects to this binary format, after which we can place them in a BLOB field.

Listing 1: Training The Model

```

1 CREATE FUNCTION train(data INTEGER, classes INTEGER,
2   n_estimators INTEGER)
3 RETURNS TABLE(classifier BLOB, estimators INTEGER)
4 LANGUAGE PYTHON
5 {
6   import pickle
7   from sklearn.ensemble
8     import RandomForestClassifier
9
10  clf = RandomForestClassifier(n_estimators)
11
12  clf.fit(data, classes)
13
14  return {'classifier': pickle.dumps(clf),
15         'estimators':n_estimators }
16 };

```

An example of a user-defined function that trains a Random Forest Classifier using Sci-Kit Learn is given in Listing 1. This is a vectorized user-defined function, and as such both data and classes are vectors of integers within the function instead of individual elements. This function can be called from within SQL with the model data, classes and the amount of estimators (i.e., model parameters) as input, and will produce a table containing the trained classifier and its meta-data as output. This table can either be stored in the database, or used directly as input to another function that uses the trained classifier (if no persistent storage is necessary). Note that it is trivial to alter this UDF to train a different model from the Sci-Kit Learn library, as all that is required is importing a different model and using that.

3.2 Classification

After the model has been trained, it is ready to accept unlabeled data and can be used to classify that data. The classification stage therefore takes as input a set of columns representing the unannotated data, and the trained classifier that will be used to classify the data. The output is the set of predicted labels produced by the classifier. Inside the user-defined function, the classifier will again have to be deserialized into an in-memory object, after which it can be used to classify the input data and produce a set of labels.

An example of a user-defined function that classifies a set of data is given in Listing 2. This function can be called from within SQL with the unlabeled data and the classifier as input, and will produce a list of predicted classes.

Listing 2: Classification

```
1 CREATE FUNCTION predict(data INTEGER, classifier BLOB)
2 RETURNS INTEGER
3 LANGUAGE PYTHON
4 {
5     import pickle
6     classifier = pickle.loads(classifier)
7     return classifier.predict(data)
8 };
```

The predict function can be used both to test a trained model and to classify a set of new data using such a model. The model can be tested by predicting a set of data for which the labels are known, and comparing the predicted labels against the new labels. The model can be used to

3.3 Ensemble Learning

In addition to only storing the trained models, we can store additional metadata about the models in the database. This metadata can include information such as parameters used to instantiate the model, or information about the effectiveness of the model obtained through testing it against certain datasets. We can then choose a model to classify new data based on this metadata, or we could classify the data using multiple models that are stored and use the results from the classifier with the highest confidence.

4 EXPERIMENTAL ANALYSIS

In this section, we demonstrate how a real classification pipeline can be integrated into a column-store database, and show how the in-database processing pipeline performs when compared against the same pipeline implemented in a standard scripting language where the input data is loaded from a file or transferred over a database socket connection.

The pipeline we use in our experiments is used to attempt to classify who people from North Carolina will vote for in the Presidential Elections based on data from the 2012 Presidential Election. For this purpose, we use two separate datasets:

- **The North Carolina Voters Dataset** contains the information about the individual voters. This is a dataset of 7.5M rows, where each row contains information about the voter. There are 96 columns in total, describing characteristics such as place of residence, gender, age and ethnicity. Note that we do not know who each person actually voted for, as this information is not publicly available.
- **The Precinct Votes Dataset** contains the aggregated voting statistics for each precinct, (i.e., how many people in each precinct voted Democrat, and how many voted Republican). This dataset has 2751 rows, one for each precinct in North Carolina.

By combining these two datasets we can attempt to classify individual voters. We know the voting records of a specific precinct, and we know in which precinct each person voted, so we can make an educated guess who each person voted for based on this information.

Preprocessing. As we do not have the true class labels for each voter, we have to generate them from the information we have about the precincts. This requires us to join the voter data with the precinct data, giving us the voting records of the precinct that each voter voted in. We then generate a “true” class label for each voter using a weighted random function based on the precinct voting records. For example, if voters in a specific precinct voted for Democrats 60% of the time, each voter in that precinct has a 60% chance of being classified as Democrat and 40% chance of being classified as Republican.

Training. After we have generated the true class labels, we have to train the model using the data and the labels. However, we don’t simply want to use all the data for training. Instead, we want to divide the data into a training set and a test set to prevent overfitting. We then feed the data in the training set to the model using the function shown in Listing 1 and store the resulting model in the database.

Testing. After the model is trained, we want to test how it performs by classifying the data in the test set and looking at the results. We can classify the voters in the test set by running the function shown in Listing 2. After having obtained the predicted class labels, we can test the accuracy of our model by comparing against the known true class labels of the data. However, since we only have the generated class labels of the individual voters, comparing the predicted labels against those would not give us a lot of information about our classification accuracy. Instead, we aggregate the total amount of predicted votes for each party by precinct. Then we compare the aggregated predictions against the known amount of votes in each precinct.

Performance Analysis. To determine how well our in-database processing solution performs compared to ad-hoc analysis pipelines we have implemented the pipeline described above both (1) using MonetDB/Python UDFs and (2) inside Python, using various different methods of initially loading the data. For loading the data in Python, we have experimented with loading from binary files (NumPy [20] files and HDF5 [18] using PyTables), CSV files using an optimized parser, transferring the data to Python through a database socket connection (with PostgreSQL [17], MySQL [21] and SQLite [3] as database servers). For the scenarios where the data is stored inside a relational database, we use SQL to perform the preprocessing steps involving joins and aggregations. Whereas for the pure Python solutions, we use the Pandas library [11] to perform these steps.

The experiments were run on a Fedora (Release 26) machine with 2.6GHz 8-core Intel Xeon processor (Turbo Boost up to 3.2GHz), 20MB shared L3 cache and 256 GB of RAM. All the tests are hot runs. The datasets and source code used for the experiments are publicly available¹.

Results. The results of the benchmark are displayed in Figure 1. The numbers display the total time required to run the entire classification pipeline, whereas the bottom gray bars indicate the time spent loading the initial data into Python and performing the initial preprocessing steps and aggregations.

¹<https://github.com/pholanda/VoterClassification>

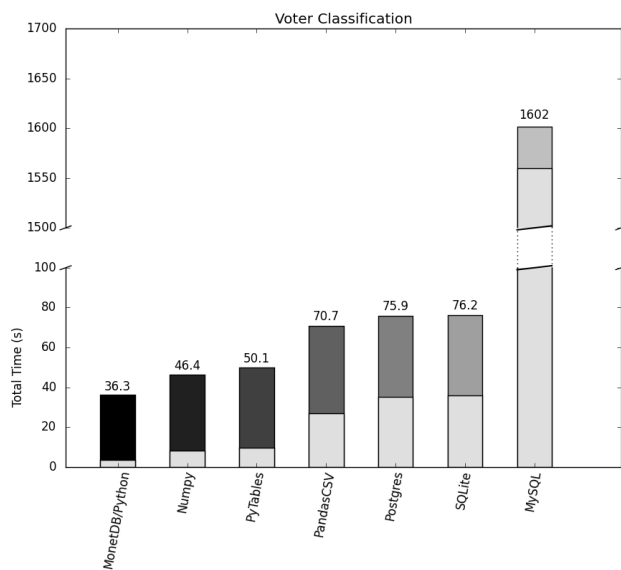


Figure 1: Voter Classification Benchmark

We can see that the in-database processing solution using MonetDB/Python is significantly faster than the alternative database solutions. The time spent on initial wrangling of the data is an order of magnitude lower than transferring it over a socket connection using the other database solutions. We also note that loading the data from CSV files is comparable in speed to transferring the data over a socket connection.

Loading the data from binary files is much faster than loading from structured text or transferring the data over a socket connection. However, this introduces additional challenges in managing the data. Especially in the case of NumPy binary files, where each of the 96 columns is stored as a separate file on disk. We do still see that the in-database processing solution spends less time on initial wrangling of the data and runs the entire pipeline significantly faster.

5 CONCLUSION

In this work, we have shown how complex analysis pipelines can be efficiently integrated into column-store databases. Using these pipelines, it is possible to perform preprocessing, training, testing and prediction using complex machine learning models directly on data stored within a relational database. We have demonstrated the efficiency gained from using these in-database processing methods, and shown the additional benefits that come with storing data in a relational database system.

5.1 Future Work

In our pipeline, there is still some unnecessary overhead in the serialization of the models. Whenever a model is stored in the database, we are serializing it to a BLOB. Before it can be used again, it must be deserialized. For larger models, this can have a performance impact. The database system could be extended to directly store snapshots of the in-memory representation of the models to avoid this (de)serialization overhead.

Additionally, we have only experimented with datasets that fit in memory. Additional work could be done on working with

out-of-memory datasets, distributed execution of the UDFs, or applying several models to the data in parallel.

ACKNOWLEDGMENTS

This work was funded by the Netherlands Organisation for Scientific Research (NWO), projects “Process Mining for Multi-Objective Online Control” (Raasveldt), “Data Mining on High-Volume Simulation Output” (Holanda) and “Capturing the Laws of Data Nature” (Mühleisen). We also would like to thank Brian Hentschel, without whom this paper would never have been written.

REFERENCES

- [1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.
- [2] R. Agrawal and K. Shim. Developing tightly-coupled data mining applications on a relational database system. In *In Proc. of the 2nd Int'l Conference on Knowledge Discovery in Databases and Data Mining*, pages 287–290. AAAI Press, 1996.
- [3] G. Allen and M. Owens. *The Definitive Guide to SQLite*. Apress, Berkely, CA, USA, 2nd edition, 2010.
- [4] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, and C. Welton. MAD skills: new analysis practices for big data. *Proceedings of the VLDB Endowment*, 2(2):1481–1492, 2009.
- [5] P. Domingos. A few useful things to know about machine learning. *Communications of the ACM*, 55(10):78–87, 2012.
- [6] X. Feng, A. Kumar, B. Recht, and C. Ré. Towards a Unified Architecture for in-RDBMS Analytics. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, SIGMOD '12*, pages 325–336, New York, NY, USA, 2012. ACM.
- [7] J. M. Hellerstein, C. RÄI, U. Wisconsin, A. Gorajek, K. Li, U. Florida, K. S. Ng, U. Wisconsin, C. Welton, D. Z. Wang, U. Florida, X. Feng, and U. Wisconsin. The MADlib analytics library, or MAD skills, the SQL.
- [8] P. Holanda, M. Raasveldt, and M. Kersten. Don't Hold My UDFs Hostage - Exporting UDFs For Debugging Purposes. In *Proceedings of the 28th International Conference on Simpósio Brasileiro de Banco de Dados, SSBDB 2017, Uberlândia, Brazil*, 2017.
- [9] S. Kandel, A. Paepcke, J. M. Hellerstein, and J. Heer. Enterprise Data Analysis and Visualization: An Interview Study. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2917–2926, Dec. 2012.
- [10] A. Kumar, M. Boehm, and J. Yang. Data Management in Machine Learning: Challenges, Techniques, and Systems. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1717–1722. ACM, 2017.
- [11] W. McKinney. Data Structures for Statistical Computing in Python. In S. van der Walt and J. Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 51 – 56, 2010.
- [12] C. Ordonez and S. K. Pitchaimalai. One-pass data mining algorithms in a DBMS with UDFs. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1217–1220. ACM, 2011.
- [13] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12(Oct):2825–2830, 2011.
- [14] M. Raasveldt and H. Mühleisen. Vectorized UDFs in Column-Stores. In *Proceedings of the 28th International Conference on Scientific and Statistical Database Management, SSDBM 2016, Budapest, Hungary, July 18-20, 2016*, pages 16:1–16:12, 2016.
- [15] M. Raasveldt and H. Mühleisen. Don't Hold My Data Hostage: A Case for Client Protocol Redesign. *Proc. VLDB Endow.*, 10(10):1022–1033, June 2017.
- [16] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with relational database systems: Alternatives and implications. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data, SIGMOD '98*, pages 343–354, New York, NY, USA, 1998. ACM.
- [17] M. Stonebraker and G. Kemnitz. The POSTGRES Next Generation Database Management System. *Commun. ACM*, 34(10):78–92, Oct. 1991.
- [18] The HDF Group. Hierarchical Data Format, version 5, 1997-NNNN. <http://www.hdfgroup.org/HDF5/>.
- [19] M. Vartak, H. Subramanyam, W.-E. Lee, S. Viswanathan, S. Husnoo, S. Madden, and M. Zaharia. ModelDB: a system for machine learning model management. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, page 14. ACM, 2016.
- [20] S. v. d. Walt, S. C. Colbert, and G. Varoquaux. The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science and Engg.*, 13(2):22–30, Mar. 2011.
- [21] M. Widenius and D. Axmark. *MySQL Reference Manual*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1st edition, 2002.

Scalable Detection of Concept Drifts on Data Streams with Parallel Adaptive Windowing

Philipp M. Grulich¹
Sebastian Breß^{1,2}

René Saitenmacher¹
Tilmann Rabl^{1,2}

Jonas Traub¹
Volker Markl^{1,2}

¹Technische Universität Berlin

²DFKI GmbH

ABSTRACT

Machine learning techniques for data stream analysis suffer from concept drifts such as changed user preferences, varying weather conditions, or economic changes. These concept drifts cause wrong predictions and lead to incorrect business decisions. Concept drift detection methods such as adaptive windowing (ADWIN) allow for adapting to concept drifts on the fly.

In this paper, we examine ADWIN in detail and point out its throughput bottlenecks. We then introduce several parallelization alternatives to address these bottlenecks. Our optimizations lead to a speedup of two orders of magnitude over the original ADWIN implementation. Thus, we explore parallel adaptive windowing to provide scalable concept detection for high-velocity data streams with millions of tuples per second.

1 INTRODUCTION

Machine learning (ML) techniques gain more and more adoption in stream analysis. They enable various use-cases such as theft detection, event classification, and failure prediction. In a nutshell, ML techniques train a ML model and apply that model when they process stream tuples (e.g., to classify them). Concept drifts cause a discrepancy between ML models and reality, which makes concept drifts a crucial challenge for machine learning on data streams. High discrepancies lead to incorrect predictions and wrong results. As a consequence, we need to continuously monitor concept drifts and retrain ML models accordingly. Stream processing engines require scalable solutions for concept drift detection and adaptation to execute ML techniques on high-velocity data streams with millions of tuples per second.

A naive approach to cope with concept drifts is to retrain the ML model periodically on a fixed size batch of recent data. This periodic re-computation is in conflict with the real-time requirement of stream processing applications for two reasons: (i) Models do not yet cover the most recent data when they are applied although the most recent data might indicate a concept drift. (ii) The data, which is reflected in the model, indicates concept drifts leading to deviations between model and reality.

In contrast to the naive solution, a wide range of approaches detects concept drifts on the fly [8]. Some monitor the evolution of different performance indicators [10, 12], while others observe the distributions on two different time-windows [9].

We study the scalability limitations of such approaches on the example of *adaptive windowing* (ADWIN) [2]. In general, ADWIN maintains a *global window* of adaptive size which is the data basis for the model computation. It trades off the variance in the *global window* (i.e., the data variance reflected in the model) against the size of the global window (i.e., the amount of data reflected by the

model). We chose ADWIN because it has proven its capabilities in a wide range of real-world applications: ADWIN was combined with Kalman Filters and demonstrated with Naïve Bayes and k-means clustering [1]. Furthermore, ADWIN is used for an online version of the Bagging method by Oza and Rusell [5] and in a parameter-free variant of the Space Saving algorithm [4]. Moreover, ADWIN is available in the open source MOA framework [3].

In this paper, we present the following contributions:

- (1) We analyze the original ADWIN approach, point out its scalability limitations, and identify its bottlenecks.
- (2) We parallelize ADWIN to overcome its bottlenecks and to provide scalable concept drift adaptation in real-time.
- (3) We evaluate the latency and throughput of our solution. It achieves two orders of magnitude speedup over the original ADWIN implementation and 54 times speedup in comparison to an optimized sequential implementation.

The source code of our implementation is available on GitHub¹.

In the remainder of this paper, we discuss ADWIN and its bottlenecks in Section 2, present our parallel adaptive windowing approaches in Section 3, and evaluate them in Section 4.

2 CONCEPT DRIFT DETECTION WITH ADAPTIVE WINDOWING

In this section, we present the original ADWIN algorithm [2] (Section 2.1), discuss an optimization based on exponential histograms (Section 2.2) and analyze the performance of the open source ADWIN implementation (Section 2.3).

2.1 The ADWIN Algorithm

ADWIN is an algorithm which detects concept drifts on the fly and adapts ML models accordingly. The algorithm maintains an adaptive window which is the basis for computing the ML model. ADWIN grows the window (i.e., adds the most recent tuples) as long as there is no concept drift detected. As a result, the model can rely on growing training data. ADWIN shrinks the window by removing old tuples when it detects a concept drift.

The algorithm does not require users to configure minimum or maximum times between concept drifts in advance because it identifies concept drifts on a per-tuple basis. This removes an important drawback of approaches which use fixed-size windows (i.e., batches) of data to recompute models periodically. In ADWIN, users configure only the confidence value $\delta \in (0, 1)$ to adjust the sensitivity of the concept drift detection.

When processing a stream tuple, ADWIN first adds the tuple to the adaptive window. Then, the algorithm analyzes the content of the adaptive window to identify concept drifts. To that end, ADWIN iterates over all possible combinations of two *large enough* sliding subwindows, as shown in Figure 1. If the value distributions of the two subwindows are *different enough*, ADWIN detects a concept drift and removes the oldest tuple from the

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

¹Source Code: <https://github.com/TU-Berlin-DIMA/parallel-ADWIN>

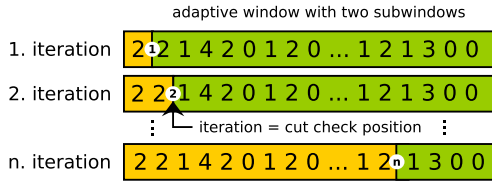


Figure 1: Iterations of the cut check procedure in ADWIN

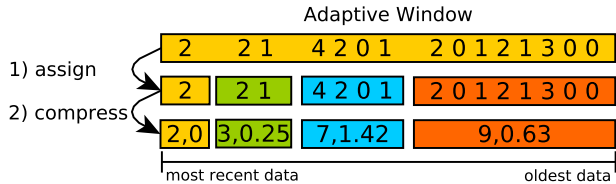


Figure 2: Exponential histogram: We first assign tuples to buckets of exponential size. We then compress buckets and store sums and variances only.

adaptive window. We call this *cut detection* because it determines when to *cut* the adaptive window in order to remove old data. The cut detection repeats removing old tuples until the content of the adaptive window does not indicate a concept drift anymore.

2.2 Exponential Histograms

A naive ADWIN implementation is computationally expensive because it performs a cut check for all possible combinations of subwindows for each tuple of the input stream. To address this problem, ADWIN uses an *exponential histogram* as underlying data structure of the adaptive window [7]. The exponential histogram assigns input tuples to buckets (Step 1 in Figure 2). Buckets of recent data contain just a few tuples. Buckets of older data contain an exponentially growing number of tuples. Each bucket stores the sum and variance of its tuples only. This reduces the memory consumption of the histogram (Step 2 in Figure 2). In summary, the number of buckets and the respective memory consumption grow logarithmically when the adaptive window grows. The cut check procedure now compares buckets instead of per-tuple subwindows which leads to an $O(\log(n))$ complexity for the cut-check procedure. Thereby, n is the number of tuples in the adaptive window and $\log(n)$ the number of buckets.

Figure 3 shows an overview of the ADWIN algorithm including exponential histograms. The algorithm performs two tasks:

- (1) ADWIN inserts arriving tuples to the adaptive window, i.e., the exponential histogram (Step 1.1 in Figure 3). This occasionally causes bucket compression and fusion of smaller buckets to larger buckets (Step 1.2 in Figure 3).
- (2) ADWIN detects concept drifts with its *cut detection* procedure and potentially removes old data from the histogram.

In the following subsection, we analyze the runtimes of the different tasks and thereby identify the bottlenecks.

2.3 Initial Performance Analysis

As a starting point of our work, we study the original ADWIN implementation, which is also part of the MOA framework [3]. This implementation is not parallel, i.e., it runs in a single thread. As we will show in detail in our experiments in Section 4, the single thread version has a low throughput compared to our parallel approaches which we present in Section 3.

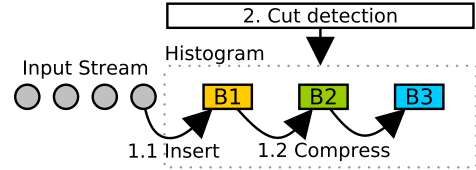


Figure 3: ADWIN overview: New tuples are added to the histogram (1.1), buckets are compressed (1.2), and the cut detection procedure identifies concept drifts (2).

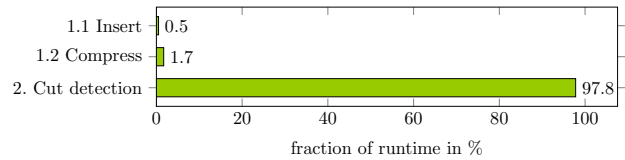


Figure 4: Runtime distribution among ADWIN tasks

In order to identify the bottlenecks of the existing implementation, we use JProfiler² and Java VisualVM³ for performance profiling. In Figure 4, we show the processing time distribution among the tasks performed by ADWIN. A single thread implementation spends about 98% of its runtime with cut checks. In comparison, the time spent on maintaining the exponential histogram is almost negligible as it contributes only 2.2% to the total processing time of a tuple. Although the number of buckets grows logarithmically with the number of tuples contained in the histogram, bucket comparisons dominate the execution effort since they are performed for each tuple. Based on the observations above, we focus our optimizations on parallelizing the cut check procedure because it exhibits the largest saving potential.

In our code analysis, we identified ways to improve the original ADWIN implementation. We did a logically equivalent reimplementa-tion to speedup the execution. The major improvement of our reimplementa-tion is the use of circular array buffers as an underlying data structure for the histogram. This reduces memory copy operations compared to the original implementation.

3 PARALLEL ADAPTIVE WINDOWING

In this section, we introduce several approaches to parallelize ADWIN in order to improve its throughput. We focus on parallelizing the *cut detection* because we identified it as bottleneck in Section 2.3. We discuss single-node parallelization in Section 3.1 and multi-node parallelization in Section 3.2.

3.1 Single-Node Parallelization

In Figure 5, we show in pseudo code how ADWIN processes an input tuple and point out possible single-node parallelizations. We present each parallelization in detail in the following subsections. First, we decouple histogram updates and cut-checks from each other such that cut-checks cannot delay processing input tuples (Section 3.1.1). This decoupling is generally applicable to algorithms that store stream statistics in a separate data structure. Then, we parallelize the cut-check procedure itself which we call *Intra Cut-Check Parallelization* (Section 3.1.2). Finally, we discuss how to perform multiple cut-check procedures in parallel in case ADWIN detects cuts. We call this *Inter Cut-Check Parallelization* (Section 3.1.3). Both parallelizations are generally applicable to all algorithms which use multidirectional iterable datastructures.

²JProfiler: <https://www.ej-technologies.com/products/jprofiler/overview.html>

³VisualVM: <http://visualvm.java.net>

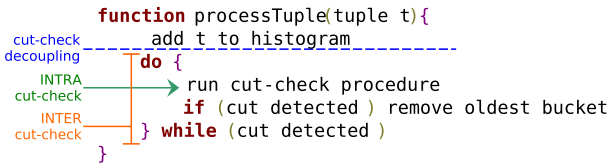


Figure 5: Scope of ADWIN parallelizations.

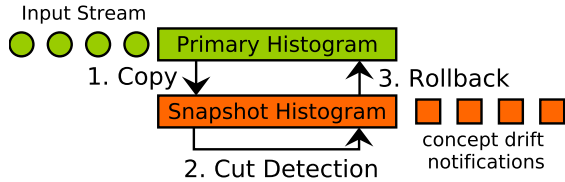


Figure 6: Optimistic ADWIN Decoupling histogram maintenance and the cut detection procedure.

3.1.1 *Cut-Check Decoupling.* We introduce an optimistic parallelization of ADWIN which assumes that most input tuples do not indicate a concept drift (i.e., a cut). This assumption regularly holds for big data streams, because real-world data follows laws of nature and causes few cuts compared to the number of tuples.

As explained in Section 2, ADWIN performs two tasks: updating the histogram with new tuples and detecting concept drifts with a cut detection procedure. Originally, ADWIN performs these tasks in succession for each tuple. This limits the throughput because the cut detection blocks the processing of the input stream.

In *Optimistic ADWIN*, we decouple histogram updates and cut checks from each other to overcome throughput limitations. The algorithm performs cut checks on a separated *snapshot histogram* and adds new tuples to a *primary histogram* concurrently. We synchronize histograms after each run of the cut check procedure.

We illustrate *Optimistic ADWIN* in Figure 6. One thread adds new input tuples to the primary histogram and a redo log. Another thread creates a deep copy of the primary histogram (step 1) and performs the cut check procedure on this copy (step 2). In case of a concept drift, we initiate a rollback which replaces the primary histogram with the snapshot manipulated by the cut detection procedure (step 3). Finally, we use the redo log to add missing input tuples to the primary histogram again. The overall process repeats continuously. Thereby, the majority of runs does not detect a cut and requires no rollback.

The main benefit of *Optimistic ADWIN* is that the cut detection procedure cannot block the input stream anymore. During one execution of the cut check procedure, new tuples are processed already. It is important to notice that *Optimistic ADWIN* introduces a latency between the insertion of a new tuple in the primary histogram and the notification about a cut. We discuss this effect in detail in Section 4. Our experiments show that we achieve detection latencies below 15 μ s with *Optimistic ADWIN*.

3.1.2 *Intra-Cut-Check Parallelization.* A single thread execution of the cut-check procedure starts to check for cuts at the ending of the histogram and moves towards the beginning (top of Figure 7). Thereby, the algorithm compares sliding subwindows as described in Section 2.1. This comparison requires sums and variances of subwindows, which are aggregates of sums and variances of buckets covered by the subwindows. The algorithm updates the aggregates of subwindow (i.e., the aggregation of buckets) incrementally when moving from one iteration to the next. As initialization for the cut-check iteration, the histogram stores and maintains its overall aggregate.

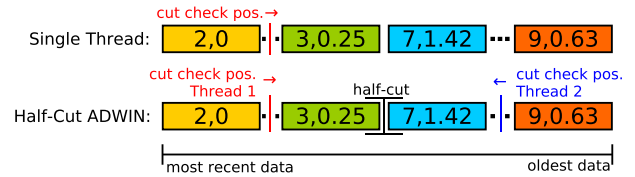


Figure 7: Half-Cut ADWIN: Parallelisation of the cut check procedure with two threads. Both threads iterate till the middle of the histogram.

We change the cut-check iteration such that two cut checks run in parallel. To that end, we introduce *Half-Cut ADWIN* in Figure 7 (bottom). Since the cut check in each iteration step is independent of cut checks of previous iteration steps, two threads can iterate over the histogram concurrently. Thereby, each thread covers half of the cut-check positions. One thread starts at the beginning and moves towards the end of the histogram and the other thread moves in the opposite direction. Both threads still update subwindow aggregates incrementally in each iteration step. *Half-Cut ADWIN* terminates the cut check procedure when both threads reach the middle of the histogram or when it finds a cut. This leads to a maximal speedup factor of two and halves the latency of the cut detection. It is also easy to add on top of existing histogram implementations.

In general, the concept of *Half-Cut ADWIN* extends to more than two threads. Therefore, the histogram needs to maintain additional aggregates of subwindows as initialization for additional threads. This overhead pays off for large histograms only. Since the number of buckets grows logarithmically, a high degree of Intra-Cut-Check Parallelism pays off seldom considering the overhead for aggregate maintenance in the histogram. *Half-Cut ADWIN* does not have this overhead because it uses the same aggregate as initialization for both threads.

3.1.3 *Inter-Cut-Check Parallelization.* If ADWIN finds a cut, it removes the oldest bucket from the histogram and repeats the cut check procedure till no further cuts are detected. This enables a *pessimistic parallelization* which assumes that we detect further cuts after removing old buckets from the histogram. While thread 1 performs cut checks on all buckets 1..n, thread 2 could already check if there will be another cut after removing an old bucket and perform cut detection on buckets 2..n. This extends to $n - 1$ parallel cut check procedures each of which could also apply *Half-Cut ADWIN*. However, the detection of cuts is usually rare compared to the total number of cut check procedures. Inter-Cut-Check Parallelization is not beneficial when we detect no cut. Respectively, we expect a minimal speedup from this approach. Still, it can reduce the latency of the cut detection, which is valuable for situations with frequent concept drifts.

3.2 Multi-Node parallelization

It is desirable to distribute stream processing applications over multiple nodes in a cluster in order to achieve linear speedup. Common distributed streaming engines such as Apache Flink [6] and Storm [11] achieve data parallelism on multiple nodes with data partitioning. Thus, each node is responsible for processing tuples of certain keys (e.g., user ids, regions, or event classes). *Half-Cut ADWIN* and *Optimistic ADWIN* are complementary to this approach and increase the throughput per partition.

It is also possible to distribute the cut detection procedure on multiple nodes. However, this requires a central shared histogram

or histogram copies with multi-version concurrency control. The coordination and network overhead for these histograms would dominate the processing in our network-bounded cluster and prevent a speedup. However, multi-node parallelization of cut checks can be beneficial if we overcome the network bottleneck with technologies such as InfiniBand.

4 EVALUATION

In this section, we provide an experimental evaluation of *Half-Cut ADWIN*, *Optimistic ADWIN*, and the state-of-the-art. We discuss our setup in Section 4.1 and present our results in Section 4.2.

4.1 Experiment Design

Throughput. The throughput of *ADWIN* depends on the size of the histogram. Large histograms contain more buckets and, thus, require more cut checks. More concept drifts (i.e., cuts) increase the throughput by reducing the histogram size. Our experiment measures the throughput for different histogram sizes.

Latency. We analyze the latency between adding a tuple to the histogram and the completion of the cut check procedure. This corresponds to the detection latency of cuts. We show the worst-case execution time of the cut check procedure which corresponds to finding a cut at the last cut check position. In addition, the latency of *Optimistic ADWIN* includes the waiting time of a tuple in the primary histogram before the decoupled cut check procedure starts. We show a range of latencies which reflects the shortest and longest possible waiting time.

Data. Since the overhead of *Optimistic ADWIN* is negligible, the performance depends on the histogram size only. Therefore, we generate batches of constant values and measure their insertion times. The insertion times include performing cut detections.

Execution Environment. We measure runtimes and latencies with *JMH*⁴ which enables reliable and reproducible microbenchmarks on a Java Virtual Machine. We run all experiments on a machine with 8GB RAM and an Intel Core i5-4210U processor.

Algorithms. We compare four versions of *ADWIN*. The open source version⁵, our optimized sequential version (Section 2.3), *Optimistic ADWIN* (Section 3.1.1), *Half-Cut ADWIN* (Section 3.1.2).

4.2 Results

Throughput. The left plot in Figure 8 shows that our sequential reimplementation is on average 5 times faster than the original implementation. *Half-Cut ADWIN* is 10 times faster than the original implementation. As expected, *Half-Cut ADWIN* is almost 2 times faster than our optimized sequential reimplementation because it reduces the execution time of cut checks by almost 50%. *Optimistic ADWIN* improves upon *Half-Cut ADWIN* by a factor of 27 and is 54 times faster than our optimized reimplementation. This leads to a 274 times speedup compared to the original *ADWIN* implementation. Moreover, *Optimistic ADWIN* decouples the insertion of tuples into the histogram from the cut check procedure. Therefore, its throughput is not directly correlated to the histogram size, which leads to a better scalability.

Latency. In the right plot in Figure 8, we show the latencies of different *ADWIN* versions depending on the histogram size. Our new *ADWIN* versions reduce latencies compared to the original implementation by up to 90% and at least by 50%. *Half-Cut ADWIN* has the lowest latency because it distributes the cut detection procedure on two threads without any snapshot and concurrency

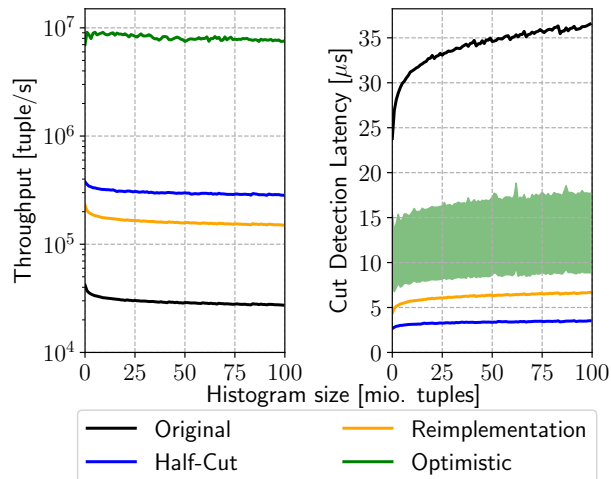


Figure 8: Throughput (left), Latency (right)

control overhead. *Optimistic ADWIN* exhibits the largest variance (10 μ s) among the latencies of different tuples. This is because of the additional waiting time between adding a tuple to the primary histogram and the start of the next cut check procedure. In the worst case, a snapshot of the histogram was taken directly before inserting the tuple. This roughly doubles the latency because we first finish the cut check procedure on the recent snapshot before we do a cut check which includes the new tuple. While *Half-Cut ADWIN* decreases the latency compared to *Optimistic ADWIN* by 70% on average, *Optimistic ADWIN* show a 27 times higher throughput. In general, all latencies are in the range of microseconds which enables fast reactions on concept drifts.

5 CONCLUSIONS

Concept drift detection, with algorithms such as *ADWIN*, is a crucial component of stream analysis. We analyzed the bottlenecks of the *ADWIN* algorithm and discussed several approaches for its parallelization. Our *Optimistic ADWIN* algorithm decouples the concept drift detection and the window maintenance. Its evaluation shows that it has two orders of magnitude higher throughput and an at least 50% lower latency than state-of-the-art solutions.

Acknowledgements: This work was supported by the EU project Streamline (688191), DFG Stratosphere (606902), and the German Ministry for Education and Research as BBDC (01IS14013A) and Software Campus (01IS12056).

REFERENCES

- [1] A. Bifet and R. Gavaldà. Kalman filters and adaptive windows for learning in data streams. In *Discovery science*, volume 4265, pages 29–40. Springer, 2006.
- [2] A. Bifet and R. Gavaldà. Learning from time-changing data with adaptive windowing. In *SDM*, pages 443–448. SIAM, 2007.
- [3] A. Bifet, G. Holmes, R. Kirkby, and B. Pfahringer. MOA: Massive online analysis. *Journal of Machine Learning Research*, 11(May):1601–1604, 2010.
- [4] A. Bifet, G. Holmes, and B. Pfahringer. Moa-tweetreader: real-time analysis in twitter streaming data. In *Discovery Science*, pages 46–60. Springer, 2011.
- [5] A. Bifet, G. Holmes, B. Pfahringer, R. Kirkby, and R. Gavaldà. New ensemble methods for evolving data streams. In *SIGKDD*, pages 139–148. ACM, 2009.
- [6] P. Carbone, A. Katsifodimos, S. Ewen, et al. Apache Flink: Stream and batch processing in a single engine. *IEEE Data Engineering Bulletin*, 36(4), 2015.
- [7] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. *SIAM journal on computing*, 31(6):1794–1813, 2002.
- [8] J. Gama, I. Žliobaitė, A. Bifet, M. Pechenizkiy, and A. Bouchachia. A survey on concept drift adaptation. *ACM Computing Surveys (CSUR)*, 46(4):44, 2014.
- [9] D. Kifer, S. Ben-David, and J. Gehrke. Detecting change in data streams. In *Vldb*, pages 180–191. VLDB Endowment, 2004.
- [10] R. Klirrenberg and I. Renz. Adaptive information filtering: Learning in the presence of concept drifts. *Learning for Text Categorization*, pages 33–40, 1998.
- [11] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, et al. Storm@twitter. In *SIGMOD*, pages 147–156. ACM, 2014.
- [12] G. Widmer and M. Kubat. Learning in the presence of concept drift and hidden contexts. *Machine learning*, 23(1):69–101, 1996.

⁴JMH - <http://openjdk.java.net/projects/code-tools/jmh/>

⁵ADWIN open source repository: <https://github.com/abifet/adwin>

Point-of-Interest Recommendation Using Heterogeneous Link Prediction

Alireza Pourali

Laboratory for Systems, Software
and Semantics (LS³)
Ryerson University
alireza.pourali@ryerson.ca

Fattane Zarrinkalam

Laboratory for Systems, Software
and Semantics (LS³)
Ryerson University
fzarrinkalam@ryerson.ca

Ebrahim Bagheri

Laboratory for Systems, Software
and Semantics (LS³)
Ryerson University
bagheri@ryerson.ca

ABSTRACT

Venue recommendation in location-based social networks is among the more important tasks that enhances user participation on the social network. Despite its importance, earlier research have shown that the accurate recommendation of appropriate venues for users is a difficult task specially given the highly sparse nature of user check-in information. In this paper, we show how a comprehensive set of user and venue related information can be methodically incorporated into a heterogeneous graph representation based on which the problem of venue recommendation can be efficiently formulated as an instance of the heterogeneous link prediction problem on the graph. We systematically compare our proposed approach with several strong baselines and show that our work, which is computationally less-intensive compared to the baselines, is able to shows improved performance in terms of precision and f-measure.

1 INTRODUCTION

The recent interest in location-based social networks (LBSNs) such as Foursquare and Gowalla has generated a high volume of user location information on the Web [17]. This has attracted researchers to analyze social data related to users' point of interests (POI) based on their preferences and personalities, which has the potential to improve the quality of higher-level applications such as civic planning, healthcare, advertising/marketing, and crime prediction [10], just to name a few. The recommendation of a point-of-interest to a given user based on her past activity is one of the applications that has already been explored by several authors [3]. These earlier works primarily use features such as spatial and temporal activity of users where both spatial features (location of venues) and temporal features (time of visit) are taken into account. The spatial features are extracted using the geographical information of the user check-ins, which are longitude and latitude of the venues. The collection of these features are then used to train classifiers to determine and recommend a point-of-interest for a given user.

In our work, we take a different perspective on the same problem of point-of-interest recommendation by formalizing user LBSN information in the form of a heterogeneous graph which consists of different node types including users, venues, venue categories, and geographical regions, among others. We propose that the problem of point-of-interest recommendation can be viewed as an instance of the link prediction problem on heterogeneous graphs. In this paper, we systematically show (1) how a collection of LBSN information including user relationships, past user-venue interaction history, venue category information and

geographical coordinates can be unified into and represented as a heterogeneous graph; (2) how meta-paths can be extracted from such a heterogeneous graph representation in order to identify potential links between users and venues (points-of-interest); and (3) that compared to much more complex baselines that incorporate spatio-temporal information into strong recommender techniques such as matrix factorization models, our proposed approach shows improved performance. We extensively compare our work with several state of the art techniques based on the gold standard dataset from [7] and show that while our heterogeneous link prediction model is quite computationally lightweight, it can offer improved performance over more computationally demanding techniques.

2 RELATED WORK

There has already been strong work on Point-of-Interest (venue) recommendation in the literature. Gamba et al. [4] have proposed an extended version of regular Markov Chains which incorporates the n previous visited venues of users. This model was then used for next location prediction using the users historical location visits. In their work, the next location of the users was predicted using the mobility Markov chain that was built for each individual user. Most of the location recommendation markov based models only use the geographical information of the locations without considering the context of the user footprints.

Matrix factorization has been widely used in location recommendation. In [2], matrix factorization is fused with geographical influence using Multi-center Gaussian Model (MGM) and social influence. In their work, location recommendation is based on the probability of a Gaussian distribution model, which is applied to the checked-in location centers along with the fusion framework with user preferences. However, the rich information of the check-in footprints such as the geographical context are not taken into account. In contrast, Yang et al. [18] have proposed a fusion framework, which exploits both spatial and temporal activity preferences (using tensor factorization) of users to predict their next point-of-interest. For each user, the spatial features are captured by building *Personal Functional Regions*, which are built based on frequented regions that the user visits. This model uses the spatial and temporal features separately for recommendation to users. In their work, each region is assigned with a category that the user is more interested in based on her historical visits. Therefore, when the user is near each region, the category assigned to that region is used for venue recommendation. Also, in this work, the lifestyle behavior of users is observed by using the temporal nature of the check-ins. It should be noted that Yang's model does not recommend venues to users and only recommends location categories.

Supervised learning models were also investigated in [9] for location recommendation. In this work, two supervised methods, linear regression and M5 trees were compared and it was

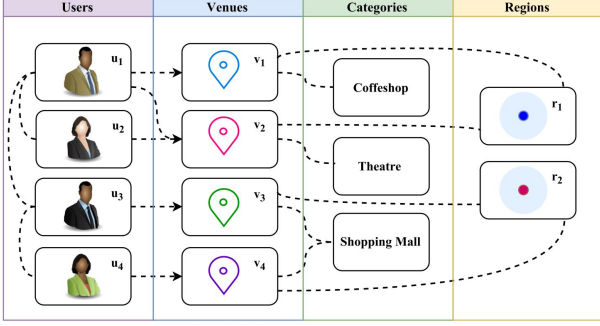


Figure 1: Sample heterogeneous graph representation of LBSN data.

found that M5 trees achieve higher prediction accuracy. Similar to other work, the set of features that were used for learning the classifiers were the user transitions between locations and spatio-temporal features of the check-in footprints. In our work, we have trained our model using the linear regression classifier and have achieved better recommendation accuracy due to using the features collaboratively and not individually compared to the work by Noulas et al. [9]. In [18] and [7], the importance of venue context information such as categories for location recommendation was also investigated. The accuracy of point-of-interest recommendations was improved by using the users' preferences that were based on the location categories they had visited. While in most existing work the current location of the user is needed for location recommendation; in our work, we show improved performance compared to the state of the art without requiring information about the last location of the user.

3 PROPOSED APPROACH

The objective of our work is to recommend a point-of-interest (venue) to a user based on her historical check-in data. Formally, given the check-in profile of a user in time interval t (Definition 1), we aim at recommending a list of venues that the user may be interested in at time interval $t + 1$.

DEFINITION 1. (User Check-in Profile). The check-in profile of user $u \in \mathbb{U}$ at time interval t , with respect to a set of venues \mathbb{V} , denoted by $CP^t(u)$, is represented by a vector of weights over the K venues, i.e., $(f_u^t(v_1), \dots, f_u^t(v_K))$, where $f_u^t(v_k)$ is equal to one if user u checked in at venue v in time interval t and 0 otherwise.

We propose to turn the problem of venue recommendation into a link prediction problem that operates over a heterogeneous graph. In addition to historical check-in data of users, there are other types of data that can be considered while recommending venues, namely venue categories, venue regions and user relationships. In this paper, we combine these data points into a unified heterogeneous graph representation model to consider them simultaneously. An illustration of our underlying representation model G can be found in Figure 1.

As illustrated in Figure 1, our representation model contains four types of nodes and four types of relations. Besides User and Venue nodes and User-Venue relations that represent historical check-in data of users, other types of data that are included consist of the following:

- **Category nodes:** In location-based social networks, venues are organized by a hierarchical category tree which provides a semantic classification of the various venues. For

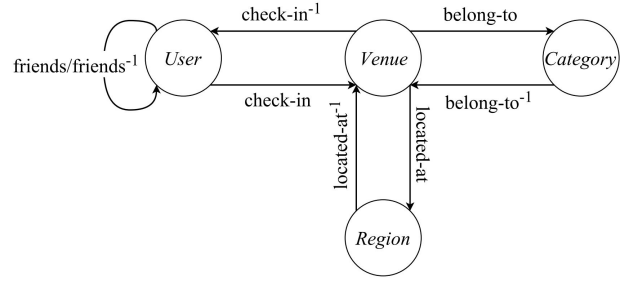


Figure 2: Network schema of the representation model.

Table 1: Meta-Paths between users and venues.

Meta-Path	Meaning of the Meta-Path
$\mathbb{U} - \mathbb{U} - \mathbb{V}$	A user visits a venue where her friends have visited
$\mathbb{U} - \mathbb{V} - \mathbb{C} - \mathbb{V}$	A user visits venues that belong to the same category
$\mathbb{U} - \mathbb{V} - \mathbb{R} - \mathbb{V}$	A user visits venues located in the same region

example, Foursquare contains a 3-level category hierarchy where categories are grouped into 10 top-level categories, such as Event, Food, Nightlife Spot and Residence. Each top-level category is classified into different subcategories. In our approach, we have infused the categories at the lowest level (Level 3) as our category nodes.

- **Region nodes:** Each region indicates a geographical area. Given the longitude and latitude of existing venues, we use X-means clustering [11] to cluster geographical coordinates and extract different regions.
- **Venue-Category relations:** Based on the hierarchical category tree defined in LBSNs for organizing venues, we assign each venue to its corresponding category in the lowest level of the hierarchy.
- **Venue-Region relation:** To identify the region of a venue, we calculate the Euclidean distance between the venue geographical coordinate (longitude and latitude) and the center of the identified regions and connect each venue to its nearest region.
- **User-User relation:** Users are connected to each other based on the friendship relation among them on the LBSN. By using this relation, potential interaction between users is also taken into account for point-of-interest recommendation.

Having built the representation model G , in order to recommend a point-of-interest to a user $u \in \mathbb{U}$, we formulate a graph-based link prediction problem that operates over G . As our representation model is a heterogeneous graph, the neighbors of an object could belong to multiple types and the paths between two objects could have different meanings. Therefore, it is not possible to apply link prediction strategies such as Adamic/Adar and Common Neighbor, which treat all types of nodes and relations as the same in the form of a homogeneous graph [6].

Sun et al. [16] proposed the concept of heterogeneous information networks and the meta-path concept for heterogeneous information network analysis, which are now widely known and used in different data mining tasks such as ranking [8], clustering

Table 2: Performance comparison of different approaches.

City	Austin			Chicago			Houston			Los Angeles			San Francisco		
	R	P	F1	R	P	F1	R	P	F1	R	P	F1	R	P	F1
Our Approach	0.056	0.096	0.071	0.096	0.130	0.111	0.207	0.122	0.153	0.120	0.112	0.116	0.072	0.116	0.089
CPOIR [7]	0.157	0.026	0.045	0.292	0.049	0.083	0.279	0.046	0.080	0.203	0.034	0.058	0.159	0.027	0.045
BasicMF	0.064	0.011	0.018	0.086	0.014	0.025	0.082	0.014	0.024	0.072	0.012	0.021	0.066	0.011	0.019
GeoCF [19]	0.122	0.020	0.035	0.227	0.038	0.065	0.165	0.027	0.047	0.164	0.027	0.047	0.126	0.021	0.036
MGMMF [2]	0.117	0.020	0.034	0.186	0.031	0.053	0.159	0.027	0.045	0.152	0.025	0.046	0.112	0.019	0.032
Markov [4]	0.086	0.014	0.025	0.116	0.019	0.033	0.102	0.017	0.029	0.096	0.016	0.027	0.088	0.015	0.025
ML [9]	0.116	0.019	0.033	0.170	0.028	0.049	0.152	0.025	0.044	0.132	0.022	0.038	0.111	0.018	0.032

[13], link prediction [1], and influence analysis [12]. In order to solve the problem of link prediction in heterogeneous graphs, Sun et al. [14] proposed the PathPredict method, i.e., meta path-based relationship prediction model to predict links between dissimilar node types. Therefore, to distinguish different types of objects and relations, following the works in [14, 15], we use the PathPredict method to determine the relevance of a point-of-interest $v \in \mathbb{V}$ for a user u . The core of the PathPredict method rests on the idea of *meta-paths*. A meta-path is a path defined over the heterogeneous network schema which can be used to define topological features with different semantic meanings. Figure 2 summarizes our representation model using a meta-structure known as the network schema.

Based on PathPredict, for the target relation $\langle \mathbb{U}, \mathbb{V} \rangle$, we define a set of meta-paths starting with type \mathbb{U} and ending with type \mathbb{V} other than the target relation itself. We extract all such meta-paths by traversing the network schema using Breadth-First Search (BFS) within a fixed length constraint ($\max = 3$). The extracted meta-paths and their semantic meaning are shown in Table 1. For example, The meta-path $\mathbb{U} - \mathbb{V} - \mathbb{C} - \mathbb{V}$, i.e., user-venue-category-venue considers those venues which belong to the same category of the historical check-in venues of a user as her next check-in venue.

Once the meta-paths are retrieved from the network schema, for each user-venue pair in the representation model G , we use the Degree-Weighted Path Count metric [5] to quantify each meta-path as a topological feature in the training step. For the given meta-path, Degree-Weighted Path Count penalizes paths, which pass through high-degree nodes. Then, given all user-venue pairs in the representation graph G and the extracted topological features for them, a logistic regression classifier is trained as the learning model to recommend a ranked list of points-of-interest for a given user.

4 EXPERIMENTS

In this section, we describe our experiments in terms of the dataset, setup and the details of the baselines used in the paper. The performance of our approach is then compared to the state of the art baselines and our findings are discussed.

4.1 Dataset and Experimental Setup

Our experiments were conducted on a dataset collected from the popular location-based social network of Gowalla introduced in [7]. It includes check-in data (longitude, latitude, timestamp, categories, among others.) of more than 600,000 users from Austin, Chicago, Houston, Los Angeles and San Francisco. For each user, we randomly select 70% of her check-ins to construct the training data and the remaining 30% of her check-ins for testing data as suggested in [7].

We compare our approach with several state of the art point-of-interest recommendation methods that are briefly described in the following:

- (1) **BasicMF** is a classical matrix factorization techniques, which only considers users’ past venue check-ins and, hence their preferences, for Point-of-Interest recommendation.
- (2) **GeoCF** [19] is based on both user preference and geographical influence which are integrated into a collaborative filtering model.
- (3) **MGMMF** [2] is a framework based on Multi-Center Gaussian Model, which combines both the user preference and MGM based check-in probability for Point of interest recommendation.
- (4) **Markov** [4] applies Mobility Markov Chain model for predicting next venue of a user based on her mobility behavior over different time intervals and the recent venues that she has visited.
- (5) **ML** [9] considers user mobility, global mobility and temporal features to describe users’ check-in behavior and applies M5 decision tree to predict the next POIs of a user.
- (6) **CPOIR** is one of the most related work in the literature by Liu et al [7], which proposes a Category-aware Point-Of-Interest Recommendation model that exploits the transition behavior of users between venue categories. They employ a matrix factorization model to predict the transition patterns of users’ interests over categories and consequently her interests in different venues.

For evaluation purposes, we measure the performance of the methods based on Precision@K, Recall@K and F1-score as suggested in [7].

4.2 Experimental Results

In this section, inline with [7], we compare the performance of our proposed approach with other state of the art baselines when Top-6 venues are recommended by each method. The results are reported in Table 2 in terms of Recall, Precision and F1-score.

It can be observed that BasicMF model, which is solely based on user interests performs worse than others for most of the cities and in terms of all three metrics. This means that incorporating other auxiliary information such as geographical, social, and temporal features leads to improved quality of venue recommendation. Markov models that incorporate temporal features outperform BasicMF; however, they perform much less accurately than the other baselines. This is because Markov models assume that a user’s mobility data is dense, as a result they may not perform so well on users’ venue data in LBSNs which is very sparse.

As another observation, MGMMF and GeoCF that fuse geographical influence and user interest into their proposed approach and take into account the correlation between these features offer more accurate recommendations compared to the ML model, which exploits geographical influence as a single feature. As mentioned before, the CPOIR model incorporates two novel features, i.e., the transition patterns of a user's interests among venues and venue categories, for the purpose of point-of-interest recommendation. It can be seen that CPOIR offers superior results compared to both MGMMF and GeoCF in all cities and in terms of all metrics. This demonstrates the benefits of these two factors to improve the quality of venue recommendations.

Inspired from these insights, in order to be able to utilize the benefits of useful features in a unified model for point-of-interest recommendation, we formalize user LBSN information in the form of a heterogeneous graph. As highlighted in Table 2, it is evident that our proposed approach outperforms all the comparison methods in all cities in terms of Precision and F1-score. This means that our approach can successfully take advantage of different features, i.e., venue categories, geographical influence, the relationship between users, and the correlation between these features to produce more accurate recommendations, i.e., less false positive. However, it can be observed that our proposed approach results in lower Recall values, i.e., more false negatives compared to others. In other words, our method is not able to identify those venues through the current limited set of defined meta-paths in this paper. To cover more meta-paths, as our future work, we intend to increase the length constraint of meta-paths, which is currently set to 3, and take into account more features in our representation model such as transition of user's interests and sentiments of user-venue related comments.

5 CONCLUDING REMARKS

In this paper, we have proposed that venue recommendation in location-based social networks can be viewed as an instance of the link prediction problem on heterogeneous graphs. As such, we have systematically shown how various types of information can be incorporated into a heterogeneous graph based on which distance metrics between nodes can be employed as features to learn ranking classifiers. We find that even a logistic regression method can effectively show competitive performance compared to the state of the art despite its simplicity and light-weight computational requirements. We further found that while our proposed approach shows improved performance over the baselines in terms of precision and f-measure, it does not show competitive performance in recall. This can be attributed to the fact that we have only employed three meta-paths and a depth of three in the BFS search.

Our future work will explore two synergistic directions: 1) We will explore whether a more extensive set of meta-paths defined over the network schema can lead to improved recall or not. We will also study whether a higher search depth is able to identify more relevant information that can be included in the link prediction process. 2) Some users provide written textual feedback about their experience at venues they visit. These include textual reviews or recommendations. We are interested in the possibility of incorporating such unstructured user feedback into the network schema to see whether textual feedback, while quite sparse, can improve the venue recommendation task.

REFERENCES

- [1] Bokai Cao, Xiangnan Kong, and Philip S. Yu. 2014. Collective Prediction of Multiple Types of Links in Heterogeneous Information Networks. In *2014 IEEE International Conference on Data Mining, ICDM 2014, Shenzhen, China, December 14-17, 2014*. 50–59. <https://doi.org/10.1109/ICDM.2014.25>
- [2] Chen Cheng, Haiqin Yang, Irwin King, and Michael R. Lyu. 2012. Fused Matrix Factorization with Geographical and Social Influence in Location-Based Social Networks. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada*. <http://www.aaai.org/ocs/index.php/AAAI/AAAI12/paper/view/4748>
- [3] Chen Cheng, Haiqin Yang, Michael R. Lyu, and Irwin King. 2013. Where You Like to Go Next: Successive Point-of-Interest Recommendation. In *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*. 2605–2611. <http://www.aaai.org/ocs/index.php/IJCAI/IJCAI13/paper/view/6633>
- [4] S. Gamba, M.O. Killijian, and M.N. Prado Cortez. 2012. Next Place Prediction using Mobility Markov Chains. In *Eurosys'13 MPM*.
- [5] Daniel S. Himmelstein and Sergio E. Baranzini. 2015. Heterogeneous Network Edge Prediction: A Data Integration Approach to Prioritize Disease-Associated Genes. *PLoS Computational Biology* 11, 7 (2015). <https://doi.org/10.1371/journal.pcbi.1004259>
- [6] David Liben-Nowell and Jon M. Kleinberg. 2003. The link prediction problem for social networks. In *Proceedings of the 2003 ACM CIKM International Conference on Information and Knowledge Management, New Orleans, Louisiana, USA, November 2-8, 2003*. 556–559. <https://doi.org/10.1145/956863.956972>
- [7] Xin Liu, Yong Liu, Karl Aberer, and Chunyan Miao. 2013. Personalized point-of-interest recommendation by mining users' preference transition. In *22nd ACM International Conference on Information and Knowledge Management, CIKM'13, San Francisco, CA, USA, October 27 - November 1, 2013*. 733–738. <https://doi.org/10.1145/2505515.2505639>
- [8] Xiaozhong Liu, Yingying Yu, Chun Guo, and Yizhou Sun. 2014. Meta-Path-Based Ranking with Pseudo Relevance Feedback on Heterogeneous Graph for Citation Recommendation. In *Proceedings of the 23rd ACM International Conference on Information and Knowledge Management, CIKM 2014, Shanghai, China, November 3-7, 2014*. 121–130. <https://doi.org/10.1145/2661829.2661965>
- [9] Anastasios Noulas, Salvatore Scellato, Neal Lathia, and Cecilia Mascolo. 2012. Mining User Mobility Features for Next Place Prediction in Location-Based Services. In *12th IEEE International Conference on Data Mining, ICDM 2012, Brussels, Belgium, December 10-13, 2012*. 1038–1043. <https://doi.org/10.1109/ICDM.2012.113>
- [10] Jon Parker, Yifang Wei, Andrew Yates, Ophir Frieder, and Nazli Goharian. 2013. A framework for detecting public health trends with Twitter. In *Advances in Social Networks Analysis and Mining 2013, ASONAM '13, Niagara, ON, Canada - August 25 - 29, 2013*. 556–563. <https://doi.org/10.1145/2492517.2492544>
- [11] Dan Pelleg and Andrew W. Moore. 2000. X-means: Extending K-means with Efficient Estimation of the Number of Clusters. In *Proceedings of the Seventeenth International Conference on Machine Learning (ICML 2000), Stanford University, Stanford, CA, USA, June 29 - July 2, 2000*. 727–734.
- [12] Chuan Shi, Yitong Li, Jiawei Zhang, Yizhou Sun, and Philip S. Yu. 2017. A Survey of Heterogeneous Information Network Analysis. *IEEE Trans. Knowl. Data Eng.* 29, 1 (2017), 17–37. <https://doi.org/10.1109/TKDE.2016.2598561>
- [13] Yizhou Sun, Charu C. Aggarwal, and Jiawei Han. 2012. Relation Strength-Aware Clustering of Heterogeneous Information Networks with Incomplete Attributes. *PVLDB* 5, 5 (2012), 394–405. http://vldb.org/pvldb/vol5/p394_yizhou_sun_vldb2012.pdf
- [14] Yizhou Sun, Rick Barber, Manish Gupta, Charu C. Aggarwal, and Jiawei Han. 2011. Co-author Relationship Prediction in Heterogeneous Bibliographic Networks. In *International Conference on Advances in Social Networks Analysis and Mining, ASONAM 2011, Kaohsiung, Taiwan, 25-27 July 2011*. 121–128. <https://doi.org/10.1109/ASONAM.2011.112>
- [15] Yizhou Sun, Jiawei Han, Charu C. Aggarwal, and Nitesh V. Chawla. 2012. When will it happen?: relationship prediction in heterogeneous information networks. In *Proceedings of the Fifth International Conference on Web Search and Web Data Mining, WSDM 2012, Seattle, WA, USA, February 8-12, 2012*. 663–672. <https://doi.org/10.1145/2124295.2124373>
- [16] Yizhou Sun, Jiawei Han, Xifeng Yan, Philip S. Yu, and Tianyi Wu. 2011. Path-Sim: Meta Path-Based Top-K Similarity Search in Heterogeneous Information Networks. *PVLDB* 4, 11 (2011), 992–1003. <http://www.vldb.org/pvldb/vol4/p992-sun.pdf>
- [17] Dingqi Yang, Daqing Zhang, Zhiyong Yu, Zhiwen Yu, and Djamel Zeghlache. 2014. SESAME: Mining User Digital Footprints for Fine-Grained Preference-Aware Social Media Search. *ACM Trans. Internet Techn.* 14, 4 (2014), 28:1–28:24. <https://doi.org/10.1145/2677209>
- [18] Dingqi Yang, Daqing Zhang, Vincent W. Zheng, and Zhiyong Yu. 2015. Modeling User Activity Preference by Leveraging User Spatial Temporal Characteristics in LBSNs. *IEEE Trans. Systems, Man, and Cybernetics: Systems* 45, 1 (2015), 129–142. <https://doi.org/10.1109/TSMC.2014.2327053>
- [19] Mao Ye, Peifeng Yin, Wang-Chien Lee, and Dik Lun Lee. 2011. Exploiting geographical influence for collaborative point-of-interest recommendation. In *Proceeding of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2011, Beijing, China, July 25-29, 2011*. 325–334. <https://doi.org/10.1145/2009916.2009962>

MetisIDX - From Adaptive to Predictive Data Indexing

Elvis Teixeira
Federal University of Ceará
Fortaleza, Brazil
elvis.teixeira@lsbd.ufc.br

Paulo Amora
Federal University of Ceará
Fortaleza, Brazil
paulo.amora@lsbd.ufc.br

Javam C. Machado
Federal University of Ceará
Fortaleza, Brazil
javam.machado@lsbd.ufc.br

ABSTRACT

Exploratory data analysis characterized by analytic query workloads over large databases is now commonplace on both academia and industry. In these scenarios, data production velocity and unknown and drifting access patterns make the choice of access methods a challenging task. In this context, adaptive indexing techniques propose the use of partial indexes that are incrementally built in response to the actual query sequence and as a byproduct of query processing to optimize the access only to the key ranges of interest. This work presents a further development to this principle by leveraging the recent query history to predict the next key ranges and index them in advance, so the queries arriving in the near future find data in its final representation and higher placed in the storage hierarchy, since data must be loaded into main memory in order to be indexed. Adaptive merging is used as base architecture for the data structures and merge operations are executed in parallel with query execution instead of being the same operation. An extreme learning machine is used to perform key range forecasting and undergo continuous training by the indexing thread. The experiments show up to 38% lower query response times over a 1000 queries than adaptive merging, therefore lower overall response times and the decoupling of indexing operations during scan executions.

1 INTRODUCTION

Important modern database applications do not have a known workload in terms of access patterns. Data subsets which are the focus of query attention change over time, and ad hoc queries should be expected. Examples of this kind of application are found in scientific work or in exploratory analysis, which is an increasingly common daily task in many business areas today. No assumptions can be made about the workload, and database systems must still be able to answer the queries from these dynamic workloads efficiently, searching through and updating suitable data structures to speed up query processing.

To accomplish this task, adaptive indexing [7] was introduced. By adapting the DBMS internal structures to quickly answer queries that follow the current trends, it enables the system to perform better according to the dynamic workload. The fundamental idea is that each time data is scanned to answer a user query, an incremental step is performed to provide an index structure or refine it, which will permit subsequent scans to prune the search space and perform better. Such operations must be simple in order to avoid adding a prohibitive overhead to query execution [2] while still being useful to speed up queries and save access to slow storage devices.

Changing database physical layout in response to the workload can be powerful if used properly. The advantages of optimizing access only for the records of interest is based on the fact that,

in most applications, a subset of the records is accessed more often than the rest. Partial indexes recognize this fact by focusing layout tuning in a subset of the indexed relation, but they lack the adaptive behavior and the possibility of incremental change. Instead of relying in periodic statistics observation, our approach continuously tracks the workload, since the access patterns and the set of records requested more frequently changes over time.

On the other hand, consider the situation in which, while trying to follow the query trends, the system organizes data in response to single queries which deviate from the underlying workload pattern. This wastes time and compromises performance. Additionally, the current query may not provide sufficient information to figure out the best key range to index in order to enable the next queries to take full advantage from the structure. More contextual information is needed. A workload model, instead of the advice from the current request, provides better guidance. These issues are similar to the problems of overfitting and generalization in pattern recognition tasks.

Improvements to adaptive indexing can be achieved by indexing key ranges not strictly equal to those of the query responses, possibly adding a stochastic component to the indexing process [4]. Another possibility is using periods of time when computing resources are not being exhaustively used to index key ranges not yet touched [10]. The main advantage of these approaches is the possibility of indexing a region of data that will be queried in the near future. When this happens, the response time will be optimal and the effort of indexing can be done independently of query processing, thus not incurring any extra overhead to it.

This work develops this analogy further by using an actual machine learning technique to create and continuously update a model of the query sequence. In other words, it leverages an adaptive structure and adds a predictive behavior to the index building operations based on a dynamic model of the workload, using the most recent requests as training data. By indexing data expected to be requested next, our index builder is less sensitive to anomalies, and avoids the effort of indexing uninteresting records. Such intelligent access structures fit naturally in the context the emerging self-driving systems [1], which promise to be able to handle highly dynamical and hybrid workloads while requiring even less manual operation than traditional systems.

The benefits of performing incremental indexing detached from query processing occurs when the forecasting succeeds and a key range is placed in the final index form before it is queried. In this case the access is logarithmic in the size of the indexed data, not on the total amount of data. If the forecasting predicts the wrong key range then the cost of the next query will be that of a scan in the partial index structure built in the first query, a partitioned B+ tree, as discussed in the next section. Even in this case, the scan will not be as costly as it would be in a strictly adaptive indexing scheme, because the scan algorithm will not have to move data around, but only to find the qualifying records. This scheme keeps the hypothesis that the workload is unknown, as in previous adaptive indexes, but it recognizes that application queries are not random, and an underlying pattern should exist.

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

2 METISIDX

MetisIDX is an indexing mechanism for relational data that targets secondary storage (HDDs, SSDs, etc) and uses the accumulated knowledge from previous queries in the workload to guide the index construction. The partitioned B+ tree, and some of the index creation and maintenance procedures are similar to those developed for adaptive merging [3]. A B-tree based structure was chosen to exploit its paged data transfer, which is naturally performed with block-addressed devices and permits meaningful data blocks to be exchanged between the levels of a memory hierarchy. However, differently from the adaptive merging approach, MetisIDX indexing routine is decoupled from data scanning and is less sensitive to workload anomalies since it leverages the information provided by multiple requests rather than the current query alone.

The learning algorithm must be lightweight enough to be executed in time frames that do not exceed the order of magnitude required to answer a single query. This constraint makes it impractical to use many state-of-art machine learning techniques such as massive deep neural networks, since using these methods would result in a training time long enough to make the resulting model already outdated, in the sense that it reflects a workload pattern that is already gone. Additionally, it must be able to update the model using new available data, new queries in this context, without discarding the previous model version altogether. A suitable technique that fulfills these requirements is the Extreme Learning Machine (ELM) [9], a class of neural networks which always has a single hidden layer, and only the weights of the connections between the hidden neurons and the output neurons must be trained.

The training data used in model updates is the sequence of key range boundaries from the last N queries processed by the system. N is a hyper parameter that has to be chosen. The trained network is the current model of the workload, used to forecast index range candidates and trigger a new merging operation. In the next training step the model is updated to keep track of possible workload trends shift. After that, the model is used to forecast the key range to be queried next, and a new merging operation is triggered. The process of training and triggering merges is carried out cyclically in a dedicated execution thread.

The first query triggers a full scan over the non-indexed data. During this first scan execution, data is read from secondary storage in chunks, called runs hereafter. Each run is then sorted using an algorithm suitable for main memory resident data. The records that belong to the query response set are then collected and returned to the user. The sorted runs are written back to disk as the leaves of a partitioned B+ tree and global ordering is achieved by introducing an artificial leading attribute whose value is the runs creation order.

The size of the runs is limited by the amount of main memory available and should be as big as possible, since longer runs provide fewer index partitions and thus faster access, because the tree must be traversed from root to leaf level for each partition. This partitioned index is already able to speed up the processing of subsequent queries, each time a search operation is executed the partitioned tree is traversed from the root to leaf level once per partition. In order to achieve optimal read access, the partitions must be merged into a single one so that the index becomes a regular B+ tree (not partitioned). In the adaptive merging [3] approach, partition merging and query processing are a single operation.

Algorithm 1: Forecast And Index Thread

Data: Predicates of last N queries
Result: Predicted query key ranges indexed continuously

```
1 while True do
2   if at least  $N$  new queries observed then
3     train neural network;
4     discard training data;
5   else
6     predict next key range;
7     if range is not locked by query thread then
8       acquire latch on key range;
9       merge key range to final index;
10      release latch;
11   end
12 end
```

In MetisIDX, two separate structures are used, one is the partitioned tree resulting from the first query, and the second, another B+ tree structure for the final index, which is composed of the results of the merging operations performed by the indexing thread. This design was chosen to minimize the efforts of structure maintenance, as a result, no merges are performed on the nodes of the partitioned tree, i. e. they are permitted to underflow and the height of the tree is fixed. In the final index, however, overflow checks happen and nodes are split as more data is moved from the partitioned tree to their final position. Adaptive merging, on the other hand, has to deal with structure maintenance for merges and splits, because the transition from the partitioned structure to the final index is accomplished by collecting the records that belong to the response set of the query and merging them into a final partition, which becomes the full index after a number of queries. All the partitions compose a single tree structure, including the final partition.

In order to answer a range query, such as the ones used in the experiment, the query processing thread traverses the partitioned tree once for each partition in the tree and collects the records of interest, then it proceeds to scan the final index to account for the case in which the required records have already been merged to that location. The predictive behavior of MetisIDX minimizes the need for the operation of scanning the partitioned tree since the merging operations are made in anticipation and eventually make entire partitions empty.

The decisions on which key ranges to merge and the actual merging operations are done independently and in parallel to query processing. Such decision process comes from the extreme learning machine that is continuously trained in background by a dedicated thread. That same thread is used to perform merges at the end of each training mini-batch. After each merging operation the indexing thread attempts to perform a new model update if enough new queries have been observed. An important difference between a predictive system and a strictly adaptive one is the fact that, by indexing a key range before it is queried, the records in the indexed range will be brought up to cache. This means that predictive indexing is also a predictive cache prefetching.

Algorithm (1) depicts this process. The whole procedure is executed in an infinite loop in the indexing thread, that continuously tries to perform merge operations if the required resources are not protected by a latch, and performs a training step if a number of queries have been observed. This number must be chosen by the user, for our setup we used 10.

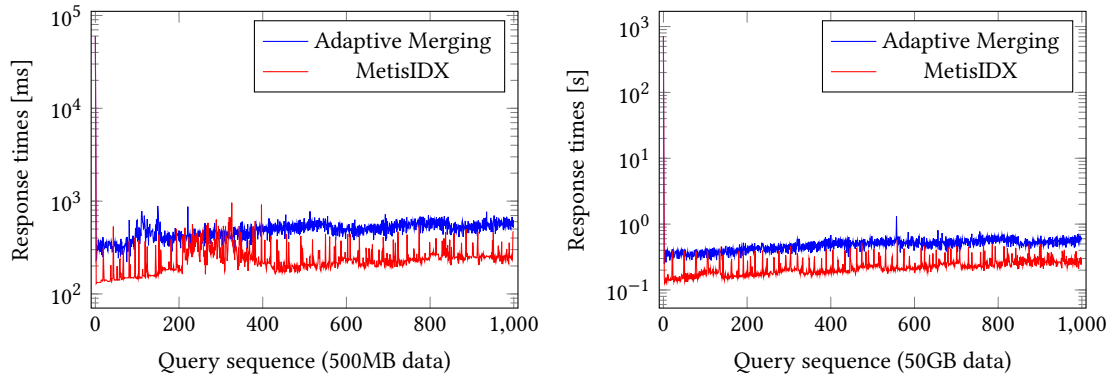


Figure 1: Response times

3 EXPERIMENTAL EVALUATION AND SETUP

MetisIDX and Adaptive Merging were both implemented for comparison in a custom storage engine named MetisDB, which implements the usual B+ tree access operations including persistence. The inclusion of adaptive merging and MetisIDX implementations in well known and complete database management systems would be a valuable way to fully evaluate and compare the performances and the implications for transaction processing by making the use of standard benchmarks possible. However, as pointed out by the authors of adaptive merging, conventional database architectures make a clear distinction between scans and index updates, treating the first as read-only, and the modifications needed to integrate the query-and-index strategies use in adaptive approaches call for whole new system architectures built with adaptive components in mind. This is the primary motivation for the MetisDB.

Since MetisIDX is not a strictly adaptive technique, and it adds indexing steps which are not part of query processing, it would make sense to compare it with Holistic Indexing, which has similar attributes. However, that technique is designed to work in the context of main memory column stores, while our approach targets tuple-based systems in secondary storage. For this reason Adaptive Merging is used as a baseline, as its application domain and data structures used are the same. The machine in which the experiments were carried out consists of a 3.1GHz Intel i5 processor, a 500GB, 7200RPM Seagate hard disk drive, and a 2x4GB DDR3 memory. The software stack is composed of a Debian GNU/Linux version 9 operating system and MetisDB was compiled using the GNU C++ compiler version 6.3.

The MetisDB storage engine contains a buffer manager that uses an LRU cache replacement policy on 8KB pages, LRU was chosen to favor recently loaded pages to remain in memory, as this is the case for recently indexed key ranges. Response times are used as a performance metric for the purpose of comparison. We use this metric instead of the usual I/O operations count used in disk-based access method evaluation because this quantity is not as directly linked to response times in MetisIDX as it is in other access methods. The reason is that, as we index data before query processing occurs and the indexing process must bring data up to the main memory cache, the select operator is expected to find its response set in the buffer pool. In other words, it does not matter how many disk accesses were performed if the data is in cache by the time one needs it.

The synthetic data used in the experiments consists of tuples with a 64bit integer used as the index key, and a 48B random string added to increase volume. Two tables were created, one containing 500MB and 7,106,208 tuples with the format described above, hereafter called $T1$, and another containing 50GB and 727,483,873 tuples, called $T2$. Two different data sizes were used to assess the effects of the domain size for the neural network predictions and cache invalidation, since the 500MB data can be entirely accommodated on cache while the larger one can not.

The query workload consists of 1000 range queries of the form `SELECT COUNT(*) ... WHERE A => QLOW AND A <= QHI`; where the key range limits, Q_{LOW} and Q_{HI} , are generated from a function Q that maps each query to a point in the search key space. Let the domain of the key be $[0, M)$ and j be the order of a query, then the Q used is

$$Q(j) = M \cdot (j/C)^2 \quad (1)$$

where M is the maximum value of the search key and C is the order of the last query, 1000 in the given setup. In this form Q will distribute the queries over the entire key range. From this function (1) Q_{LOW} and Q_{HI} are derived by

$$\begin{aligned} Q_{LOW}(j) &= Q(j) - R_1 \\ Q_{HI}(j) &= Q(j) + R_2 \end{aligned} \quad (2)$$

where R_1 and R_2 in (2) are random positive values generated by a normal distribution with standard deviation equal to 1% of the key range. This parameter determines the selectivity of the queries and additional executions with different values were performed with similar results. These are the functions the neural networks learn. A quadratic function was chosen to define the access pattern as it is a simple non-linear function, and the goal is not to test the ELM forecasting capabilities for complex functions as it is done elsewhere [5]. This quadratic function, even though non-linear, is a sequential access, the worst case for adaptive indexes since, as a key range is never queried more than once, each one faces the non-indexed part of the data.

Two ELMs were used, one learns Q_{LOW} and the other learns Q_{HI} , both as functions of j . Each network has one neuron in the input and output layers, since the target function is one-dimensional, and four neurons in the hidden layer. A test with 4 neurons was carried out and yielded the same results, since 4 is enough for such simple function. In a real application where the access pattern may be more complicated, the use of more neurons in the hidden layer is advisable.

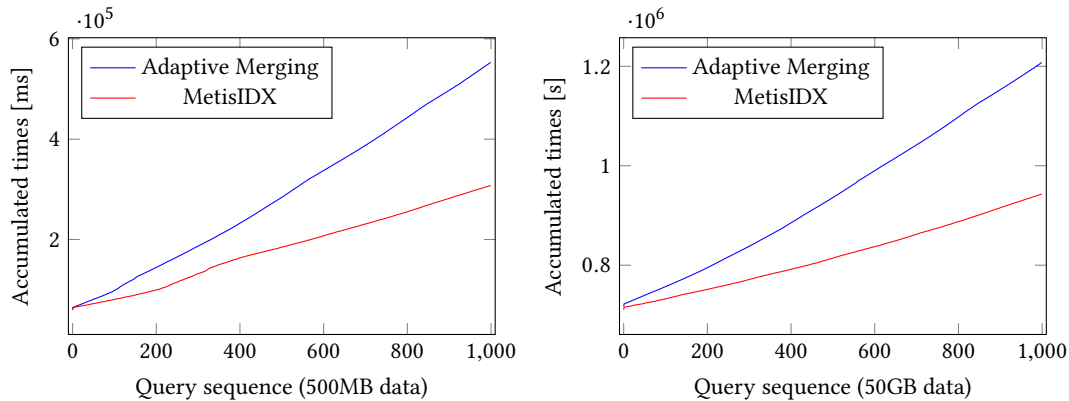


Figure 2: Accumulated response times

Figure (1) shows the response times of MetisIDX and adaptive merging for the two data sizes. For both experiments the fact that the first query has a cost much higher than the ones that follow is visible, this result is compatible with the related work in adaptive indexing in general. The queries over the 500MB data have a noisy behavior likely due to the CPU usage of the training thread and other processes running in parallel, since the data is small enough to be all in memory after the first query. On the other hand, the queries over 50GB data have a more consistent behaviour and the advantage of decoupling indexing from query processing becomes clear. The soft increase seen along the query sequence is due to the growing number of cached pages, which increases the addressing costs.

Figure 2 Shows the accumulated query processing time, that is, the time spent to perform the first N queries as a function of N . The behavior of the curves show that as more queries are processed the gain in overall performance increases so that long running applications benefit the most from the technique. This also points a common trait of adaptive indexes, that is, as more queries are processed, the system gains knowledge of the data stored and of the workload.

In a few queries, the response times of MetisIDX jumps to the level of adaptive indexing. This happens when a key range starts being indexed and then is requested by a query before the indexing action finishes. It is the only situation in which the query processing thread waits. Since the observation window used is 10 queries and the indexing action for the next immediate query happens after each observation window, this is expected to be the periodicity of these peaks.

4 CONCLUSIONS AND FUTURE WORK

MetisIDX is able to speed up access by range queries by decoupling index building efforts from query processing while maintaining the workload oriented behavior of adaptive indexes. In this technique not only the current query is treated as a hint on how to physically organize the data but the entire query sequence is taken to be a sample of the underlying workload patterns. The workload is still assumed to be unknown, only the existence of an underlying pattern is required.

It shares the concurrency concerns of other adaptive techniques in terms of application access. Additionally, indexing occurs in parallel to query processing when the key ranges do not overlap. These considerations call for a latch free alternative to data structure access in order to alleviate latch contention, or

even make the two actions race free, and increase throughput. An option to address these challenges is the adaptation of the approach presented here to the context a latch-free B-tree based structure, such as the Bw-tree [8] and a multiversion strategy to distinguish the data accessed by the current query and that being indexed may also be an option.

An in-depth analysis of the overall concurrency issues would be valuable not only for this particular technique but for any machine learning and pattern recognition based access methods. These are interesting issues for big data exploration as building full indexes upfront is an increasingly less attractive option [6] as data volumes grow. Learning from the data not only about the information it carries but also about the best ways to access it is a reasonable and, as far as we know, open research problem.

Acknowledgements

This research was partially supported by FUNCAP/CE-Brazil and LSBD/UFC.

REFERENCES

- [1] J. Arulraj, A. Pavlo, and P. Menon. Bridging the archipelago between row-stores and column-stores for hybrid workloads. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 583–598, 2016.
- [2] M. Athanassoulis, M. S. Kester, L. M. Maas, R. Stoica, S. Idreos, A. Ailamaki, and M. Callaghan. Designing access methods: The RUM conjecture. In *Proceedings of the 19th International Conference on Extending Database Technology, EDBT 2016, Bordeaux, France, March 15-16, 2016, Bordeaux, France, March 15-16, 2016*, pages 461–466, 2016.
- [3] G. Graefe and H. A. Kuno. Self-selecting, self-tuning, incrementally optimized indexes. In *EDBT 2010, 13th International Conference on Extending Database Technology, Lausanne, Switzerland, March 22-26, 2010, Proceedings*, pages 371–381, 2010.
- [4] F. Halim, S. Idreos, P. Karras, and R. H. C. Yap. Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores. *PVLDB*, 5(6):502–513, 2012.
- [5] G. Huang, Q. Zhu, and C. K. Siew. Extreme learning machine: Theory and applications. *Neurocomputing*, 70(1-3):489–501, 2006.
- [6] S. Idreos, S. Manegold, H. A. Kuno, and G. Graefe. Merging what’s cracked, cracking what’s merged: Adaptive indexing in main-memory column-stores. *PVLDB*, 4(9):585–597, 2011.
- [7] M. L. Kersten and S. Manegold. Cracking the database store. In *CIDR*, pages 213–224, 2005.
- [8] J. J. Levandoski and S. Sengupta. The bw-tree: A latch-free b-tree for log-structured flash storage. *IEEE Data Eng. Bull.*, 36(2):56–62, 2013.
- [9] N. Liang, G. Huang, P. Saratchandran, and N. Sundararajan. A fast and accurate online sequential learning algorithm for feedforward networks. *IEEE Trans. Neural Networks*, 17(6):1411–1423, 2006.
- [10] E. Petraki, S. Idreos, and S. Manegold. Holistic indexing in main-memory column-stores. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1153–1166, 2015.

Efficient SIMD Vectorization for Hashing in OpenCL

Tobias Behrens¹ Viktor Rosenfeld¹ Jonas Traub² Sebastian Breß^{1,2} Volker Markl^{1,2}
¹DFKI GmbH ²Technische Universität Berlin

ABSTRACT

Hashing is at the core of many efficient database operators such as hash-based joins and aggregations. Vectorization is a technique that uses *Single Instruction Multiple Data* (SIMD) instructions to process multiple data elements at once. Applying vectorization to hash tables results in promising speedups for build and probe operations. However, vectorization typically requires intrinsics – low-level APIs in which functions map to processor-specific SIMD instructions. Intrinsics are specific to a processor architecture and result in complex and difficult to maintain code.

OpenCL is a parallel programming framework which provides a higher abstraction level than intrinsics and is portable to different processors. Thus, OpenCL avoids processor dependencies, which results in improved code maintainability. In this paper, we add efficient, vectorized hashing primitives to OpenCL. Our results show that OpenCL-based vectorization is competitive to intrinsics on CPUs but not on Xeon Phi coprocessors.

1 INTRODUCTION

Modern processors support *Single Instruction Multiple Data* (SIMD) extensions. These *vectorized instructions* process multiple data values in a single instruction to increase the computational efficiency of a program. Database operators that use SIMD instructions are several times faster than scalar operators, because they process multiple tuples at once [8, 10, 11, 13].

Compilers expose SIMD instructions through function-like primitives called *intrinsics* [5]. Since intrinsics correspond directly to SIMD instructions of a processor, they are processor-dependent. Different processor architectures use specific instruction sets and each processor generation typically adds new instructions. Consequently, supporting vectorized database operators on different processors requires continuous maintenance of a growing code base and increases development costs.

Parallel computing frameworks such as OpenCL abstract from low level intrinsics and enable programmers to write code in a restricted dialect of C. The major advantage of OpenCL is its portability. Processor-specific compilers translate OpenCL programs to efficient machine code. OpenCL natively supports vectorized data types, which are directly compiled to the native SIMD instructions of a particular processor. However, OpenCL’s vectorized instruction set is limited to arithmetic, logical, and permutation operations. Therefore, we need to emulate more complex SIMD instructions such as *Gather* and *Scatter* [8].

In this paper, we leverage OpenCL to provide vectorized implementations of database operators which are portable to different instruction set architectures and processors (e.g., Intel CPUs and Xeon Phi coprocessors). OpenCL programs are implemented in special functions called *kernels*. We provide vectorized kernels for the data movement primitives *selective load*, *selective store*, *gather*, and *scatter* [8]. These primitives are essential building blocks of hash-based operators. We use vectorized hashing operations for

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

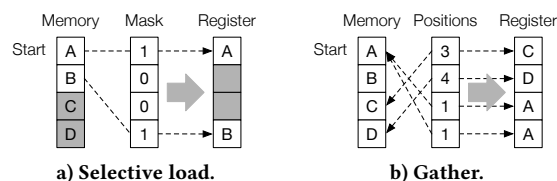


Figure 1: Vectorized data movement primitives. Grey boxes indicate values that are neither read nor written.

our case study because they are at the core of many database operators. Our results show that portable OpenCL-based hashing is competitive to processor-specific vectorized implementations.

Specifically, we make the following contributions:

- (1) We adapt vectorized data movement primitives to the OpenCL computation model. Using these primitives, we formulate explicitly vectorized algorithms of different data processing operations (Section 3).¹
- (2) We compare our OpenCL-based approach with intrinsics-based SIMD instruction sets – namely, AVX2 on a Haswell CPU and AVX512 on a Xeon Phi coprocessor (Section 4).

2 BACKGROUND

2.1 Vectorized Data Movement Primitives

Vectorized data movement primitives move data between *SIMD lanes* (i.e., the components of SIMD registers) and memory locations [8]. *Selective Load*, *Selective Store*, *Gather*, and *Scatter* are such data movement primitives. *Selective Load* (Figure 1a) selects data from *contiguous* memory (starting at an offset) and copies it into SIMD lanes specified by a bitmask. *Selective Store* is the inverse operation of *Selective Load*, which copies data from SIMD lanes into *contiguous* memory. *Gather* (Figure 1b) selects data from *discontiguous* memory and copies it into SIMD lanes. A separate SIMD register provides the pointers to data elements. *Scatter* is the inverse operations of *Gather* which copies data from SIMD lanes to *discontiguous* memory. Modern processors support these operations natively to a certain extent: The Intel Xeon Phi coprocessor, which uses the AVX512 SIMD instruction set, supports all four primitives. Intel Haswell CPUs, which use the AVX2 SIMD instruction set, support *Gather* operations only. However, Polychroniou et al. emulate these primitives using basic SIMD permutation instructions at a small performance penalty [8].

2.2 Vectorized Linear Probing in Hash Tables

A probe operation iterates over many keys. Vectorized hash tables use a SIMD register (k) to probe multiple keys (k_i) at once. We show the initial iteration step of vectorized hashing in Figure 2a:

- ① We load probe keys (k_i) into a SIMD register (k) with *Selective Load*. In the first iteration, we load all SIMD lanes as indicated by the green bitmask.
- ② For each probe key k_i in the SIMD register, we compute the hash h_i and store it in a SIMD register h . We keep separate SIMD registers for probe keys (k) and their hashes (h).
- ③ We use the hash values as position pointers in a *Gather* operation to load buckets from the hash table. We store the found keys in a new SIMD register k' .

¹Source Code: <https://github.com/TU-Berlin-DIMA/OpenCL-SIMD-hashing>

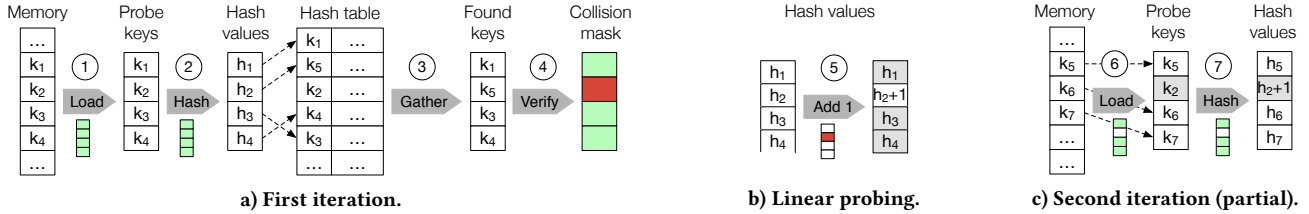


Figure 2: Vectorized operations on a linear probing hash table.

- ④ We compare the original probe keys (k) with the keys we retrieved from the hash table (k'). This comparison results in a collision mask (c). Light green boxes in the mask indicate matches and dark red boxes indicate collisions.

In our example, we find three expected keys: k_1 , k_3 , and k_4 . However, the hash table contains the key k_5 at position h_2 instead of k_2 , which indicates a collision (i.e., k_2 and k_5 have the same hash). In general, there are three possible cases per probe key: (1) The bucket contains the key. (2) The hash bucket is empty. (3) The key in the hash bucket and the probe key are different.

In cases one and two, we replace the matched probe key in k with a new probe key. In case three, we keep the probe key k_i , but increment its hash value h_i in h to probe the next bucket (Figure 2b, step ⑤). We now continue with the next iteration.

- ⑥ We use the collision mask to load new keys into the SIMD lanes for which there was no collision in the previous iteration, leaving key k_2 unchanged as described above.
- ⑦ We compute hashes for the new keys in k , leaving the value h_2+1 unchanged. Steps ③ to ⑦ repeat until all probe keys are processed.

Note that we need to write the payload (matched keys) into an output buffer with *Selective Store* between steps ④ and ⑤, e.g., to perform a hash join. For simplicity, we ignore empty hash buckets in the illustration in Figure 2. To handle empty hash buckets, we need to compute three bitmasks. The first bitmap (c') indicates found keys, the second bitmap (c'') indicates empty buckets, and the third bitmap (c) indicates collisions ($c = \neg c' \wedge \neg c''$).

If we are building a hash table, instead of using a *Gather* operation to load payloads, we use a *Scatter* operation to store keys. Since multiple SIMD lanes can point to the same hash bucket, we need to verify afterwards if the hash table contains the expected keys using steps ③ and ④. For successfully stored keys, we store the payload in the hash table using a scatter operation. For conflicting keys we need to probe the next bucket using step ⑤.

Note that the hash table is unaware of being accessed with vectorized operations. We can use the described scheme to access hash tables built with scalar operations and vice versa.

2.3 Related Work

Heimel et al. showed that OpenCL is a viable way to run database systems on heterogeneous processors [4]. Our work complements this research by porting vectorization optimizations to OpenCL. Pirk et al. introduced Voodoo – a vector algebra that abstracts from the underlying processor and generates OpenCL code [7]. Breß et al. introduced Hawk – a hardware-tailored code generator which produces custom code for heterogeneous processors [3]. Our work complements Voodoo and Hawk with templates for efficient vectorized hash tables in OpenCL.

Richter et al. showed a seven-dimensional analysis of hash tables [9]. Balkesen et al. [1] and Blanas et al. [2] studied efficient hash joins focusing on radix joins. Jha et al. optimized hash joins for Xeon Phi, but with limited use of SIMD instructions [6].

Zhou and Ross introduced vectorizations for major database operators (selections, joins, aggregations, etc.) [13]. They pointed out opportunities of SIMD in databases but did not apply SIMD to hash tables. Complementary to our work, Ye et al. evaluated different strategies for efficient aggregations on multi core CPUs [12].

3 PORTABLE VECTORIZED HASHING

In this section, we review vectorization support in OpenCL and present the internals of our OpenCL-based primitives *Selective Load*, *Selective Store*, *Gather*, and *Scatter*.

3.1 Vectorization Support in OpenCL

To implement data movement primitives, we use several built-in functions. OpenCL natively supports vector data types which represent SIMD registers. OpenCL also provides arithmetic and logical operations on vector types. For example, we compare two vectors containing four values in Listing 1. We can also access individual vector components by their indices which increase from left to right. For example, `probeKeys.s0` selects the left-most component containing the value k_1 .

```

1 uint4 probeKeys = {k1, k2, k3, k4};
2 uint4 foundKeys = {k1, k5, k3, k4};
3 uint4 mask = probeKeys == foundKeys; // {-1, 0, -1, -1}

```

Listing 1: Vectorized data types and operations in OpenCL.

The function `shuffle(input, mask)` returns a vector in which each component s_i contains the value of `input.sj` that is specified by the corresponding component s_i in `mask`, i.e., $j = \text{mask}.s_i$. The function `select(a, b, mask)` returns a vector in which each component s_i contains the value of `a.si` if `mask.si ≥ 0` and `b.si` otherwise.

3.2 Implementation of Primitives

Selective Load. Listing 2 shows the internals of the *Selective Load* primitive which we introduce with the other primitives in Section 2.1 (Figure 2a). The algorithm has four parameters: (1) `input`: source memory buffer, (2) `offset`: read offset on `input`. (3) `vector`: target vector, and (4) `mask`: indicates the components in `vector` which will be overwritten. The algorithm uses the parameters as follows: (1) It moves components which will be overwritten to the left of the target vector and adjusts the mask accordingly (Lines 3–6). (2) The algorithm loads the input data into a temporary vector (Line 7). (3) It copies the left-most values from the temporary vector into the target vector according to the mask (Line 8). (4) The algorithm moves the components of the target vector back to their original positions (Lines 11–12).

The shuffle functions used in steps 1 and 4 of the algorithm require permutation masks (left and back) to reorder the target vector. To speed up execution, we precompute these masks and store them in two lookup tables (one per step).

```

1 // Inputs: input, offset, vector, mask
2 // Outputs: offset, vector
3 ushort index =  $\sum_{i=0}^n -2^{n-i} \times \text{mask}.s_i$ ;
4 uchar8 left = convert_uint8(move_left_masks[index]);
5 vector = shuffle(vector, left);
6 int8 mask2 = shuffle(mask, left);
7 uint8 tmp = vload8(0, &input[offset]);
8 vector = select(vector, tmp, mask == -1);
9 offset += popcount(index);
10 // lines below can be omitted for optimization
11 uchar8 back = convert_uint8(move_back_masks[index]);
12 vector = shuffle(vector, back);

```

Listing 2: OpenCL implementation of *Selective Load*.

```

1 // Inputs: output, offset, vector, mask
2 // Outputs: offset
3 uchar index =  $\sum_{i=0}^n -2^{n-i} \times \text{mask}.s_i$ ;
4 uchar8 left = convert_uint8(move_left_masks[index]);
5 uint8 tmp = shuffle(vector, left);
6 vstore8(tmp, 0, &output[offset]);
7 offset += popcount(index);

```

Listing 3: OpenCL implementation of *Selective Store*.

```

1 // Inputs: input, vector, mask
2 // Output: vector
3 vector.s0 = input[mask.s0];
4 vector.s1 = input[mask.s1];
5 // ... up to vector.s7 = input[mask.s7];

```

Listing 4: OpenCL implementation of *Gather*.

We select the required permutation mask depending on the mask parameter (Line 3). The lookup tables together consume 4 KB and fit comfortably in the L1 cache.

Step 4 of the algorithm is only required if the calling code expects the components of target vector to remain in the original order. In many cases the calling code does not have this expectation. For example, the hashing scheme in Section 2.2 does not require the original order as long as reordering is mirrored between probe keys and the collision mask. Therefore, we optimize the general implementation shown above by omitting step 4 of the algorithm (Lines 11 and 12). This optimization discards one lookup table and saves the respective space in the L1 cache.

Selective store. The implementation of Selective Store is provided in Listing 3. It is similar to Selective Load and has the same parameters. Again, we move the components of vector that will be stored according to mask to the left (line 5). The values are then written to output (Line 6). Since the original vector is unchanged, we do not have to shuffle it back.

Gather and Scatter. We provide the implementations for the Gather and Scatter primitives in Listings 4 and 5. We access the components of vector and mask by their indices (see Section 3.1). Overall, we replace a complex and non-portable implementation based on intrinsics (Listing 6) with a fairly simple and portable implementation in OpenCL. Our implementation offers the same functionality, while improving maintainability and portability.

4 EVALUATION

4.1 Experimental Setup

Execution Environment. We evaluate our implementation on two processors, an Intel Core i7-6700K CPU² and a Xeon Phi 7120P coprocessor³ based on Intel’s MIC architecture. As of writing, the newer Xeon Phi Knights Landing (KNL) product line does not yet support OpenCL. On the Xeon Phi, we utilize 60 threads for our measurements to emphasize call overheads. On the CPU,

²4 GHz, 4 physical cores, 2 threads/core, 8 MB L3 shared, 32 GB RAM

³1.24 GHz, 61 physical cores, 4 threads/core, 512 kB L2 per-core, 16 GB RAM

```

1 // Inputs: input, vector, mask
2 output[mask.s0] = vector.s0;
3 output[mask.s1] = vector.s1;
4 // ... output[mask.s7] = vector.s7

```

Listing 5: OpenCL implementation of *Scatter*.

```

1 // Inputs: uint64_t* table, __m128i index
2 __m128i index_R = _mm_shuffle_epi32(index, _MM_SHUFFLE
3     ↪ (1, 0, 3, 2));
4 __m128i i12 = _mm_cvtepi32_epi64(index);
5 __m128i i34 = _mm_cvtepi32_epi64(index_R);
6 size_t i1 = _mm_cvtsi128_si64(i12);
7 size_t i3 = _mm_cvtsi128_si64(i34);
8 __m128i d1 = _mm_loadl_epi64((__m128i *)&table[i1]);
9 __m128i d3 = _mm_loadl_epi64((__m128i *)&table[i3]);
10 i12 = _mm_srli_si128(i12, 8);
11 i34 = _mm_srli_si128(i34, 8);
12 size_t i2 = _mm_cvtsi128_si64(i12);
13 size_t i4 = _mm_cvtsi128_si64(i34);
14 __m128i d2 = _mm_loadl_epi64((__m128i *)&table[i2]);
15 __m128i d4 = _mm_loadl_epi64((__m128i *)&table[i4]);
16 __m256i d12 = _mm256_castsi128_si256(_mm_unpacklo_epi64
17     ↪ (d1, d2));
18 __m256i d34 = _mm256_castsi128_si256(_mm_unpacklo_epi64
19     ↪ (d3, d4));
20 __m256i res = _mm256_permute2x128_si256(d12, d34,
21     ↪ _MM_SHUFFLE(0, 2, 0, 0));

```

Listing 6: SIMD implementation of *Gather* [8].

we utilize eight threads to emphasize attainable throughput. We use native implementations based on intrinsics [8] as a baseline. First, we perform microbenchmarks to measure the performance of our vectorized data movement primitives in isolation. We then evaluate the complete hash table implementation by executing a hash join. We separately measure building the hash table on the inner join table, and probing it with keys from the outer table.

Every experiment is executed 20 times. We prevent autovectorization of the scalar and intrinsics implementation. For each experiment we generate new test data to obtain unbiased results.

Load and Store. To evaluate both primitives, we stream 100 million keys (10^8 , 32-bit int) from memory into a SIMD register (or vice versa) and measure the throughput. For each invocation, we change the mask indicating which SIMD lanes are accessed, accessing four out of eight lanes on average. The large number of keys simulates a large outer table of a hash join.

Gather and Scatter. To evaluate Gather and Scatter, we read 100 million keys from discontinuous memory locations into a SIMD register. We use memory regions of different sizes, from 4 kB to 64 MB, to simulate hash tables built on inner tables used in a hash join. The stride size between the memory locations depends on the size of the memory region. Especially for large memory regions, we chose the indices so that the accessed data does not fit into the processor cache.

Hash Join Build and Probe. In general, we adopt the experimental setup of Polychroniou et al. [8] in order to obtain comparable results. We build a hash table on the inner table with a load factor of 50% and evaluate hash tables of different sizes, from 4 kB to 64 MB. On the Xeon Phi, we cannot build 60 hash tables of 64 MB in OpenCL due to OpenCL memory allocation restrictions. As the reference [8] does not provide an intrinsics implementation for the build on CPUs, we omit the curve.

To simplify partitioning the workload to different threads, the number of keys in the outer table depends on the processor. On the CPU, the outer table contains 100 million keys. On the Xeon Phi, it contains 245.76 million keys. We chose the keys in the outer table so that on average every tenth key is found in the hash table. Note, that our hash join implementation includes writing the matched probe keys to an output buffer.

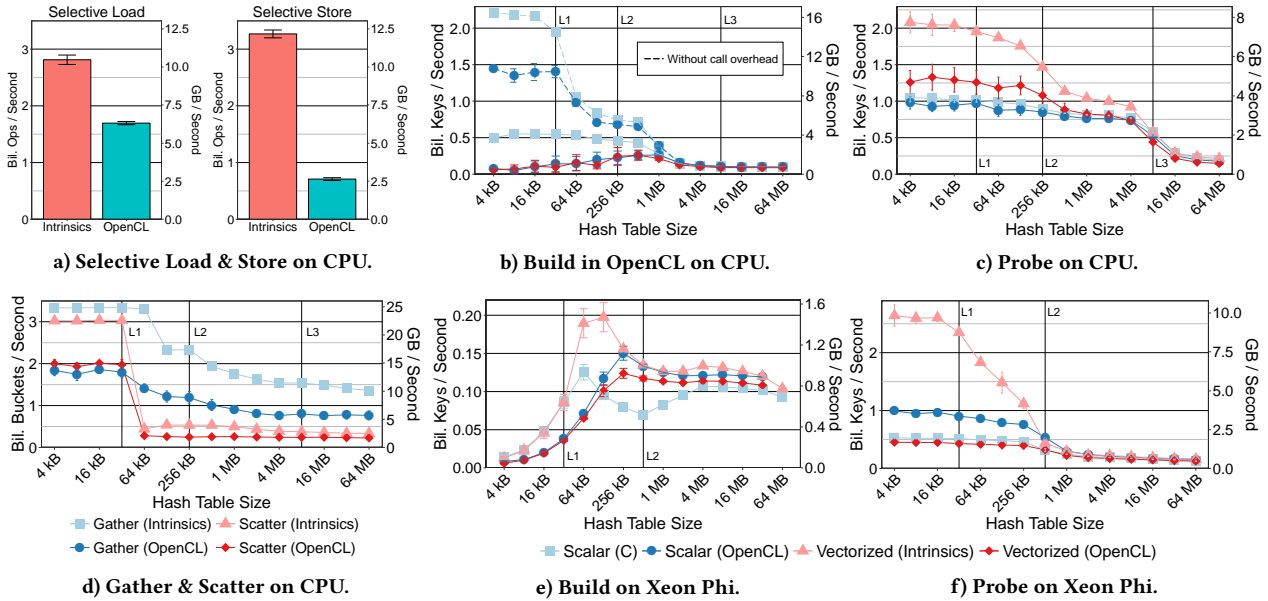


Figure 3: Evaluation results on the Intel Core i7-6700K CPU and Xeon Phi 7120 coprocessor.

4.2 Results

Selective Load and Store. Figure 3a shows the results of the Selective Load and Store microbenchmarks. The intrinsic-based version of Selective Load outperforms the portable OpenCL implementation by a factor of 1.75. The native Selective Store implementation is 4.6 times faster than the OpenCL version.

Gather and Scatter. Figure 3d shows the results of the Gather and Scatter microbenchmark. If the hash table fits into the L1 cache, the intrinsic implementation of Scatter is 1.5 times faster than the OpenCL version. However, if the hash table size exceeds the L1 cache, the performance of both implementations drops. The native implementations of Gather is between 1.75 and 1.9 times faster than the OpenCL-based implementation.

Hash Join Build. We present the results of the hash build experiment for the CPU in Figure 3b and for the Xeon Phi in Figure 3e. On the CPU, the OpenCL-based vectorized implementation is marginally slower than the OpenCL-based scalar implementation. Both are significantly slower than the C-based build and exhibit a curious rising trend up to a hash table size of 1 MB. This trend is due to overheads of OpenCL kernel invocations. To illustrate this, we also show scalar implementations which build 10000 hash tables inside a single function to minimize call overhead (dashed lines). These curves follow the expected shape. On the Xeon Phi, all implementations show a rising trend due to function call overhead. These overheads cause the bowl-shaped form of the curves which is most visible for the C-based scalar implementation. However, we cannot fully explain why the curves of the C-based and OpenCL-based scalar implementations cross between the hash table sizes of 64 kB and 128 kB.

Hash Join Probe. Figures 3c and 3f show the result of the hash probe experiment on the CPU and the Xeon Phi. We compare the vectorized OpenCL-based implementation with an intrinsic-based implementation [8] and a scalar implementation. On the CPU, both vectorized implementations outperform the scalar version as long as the hash table fits into L2 cache. For 4 kB hash tables, the intrinsic-based implementation is twice as fast as the scalar implementation, whereas the OpenCL-based implementation is 1.3 times faster. For small hash tables on the Xeon Phi, the intrinsic-based implementation greatly outperforms the scalar version whereas the OpenCL-based vectorized implementation

is slower. On this processor, the data movement primitives are implemented directly as SIMD instructions which perform much faster than implementations that rely on an emulation.

5 CONCLUSION

Vectorized database operators improve performance but require processor-specific APIs. In this paper, we vectorize the essential primitives *Gather*, *Scatter*, *Selective Load* and *Selective Store* in OpenCL to reduce code complexity and to ensure portability.

We conduct an evaluation on CPUs and Xeon Phi coprocessors. In general, vectorized hashing based on intrinsic outperforms OpenCL-based hashing. Hash tables usually exceed processor caches. In this case, both variants are memory-bound and perform similarly. However, on CPUs, OpenCL-based vectorized hashing outperforms scalar hashing for moderately sized hash tables that fit into the L2 cache. In this case, our OpenCL-based hashing scheme is competitive to intrinsic-based hashing.

Acknowledgements: This work was funded by the EU projects SAGE (671500) and E2Data (780245), DFG Stratosphere (606902), and the German Ministry for Education and Research as BBDC (01IS14013A) and Software Campus (01IS12056).

REFERENCES

- [1] Cagri Balkesen, Jens Teubner, et al. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *IEEE ICDE*. 362–373.
- [2] Spyros Blanas, Yanan Li, et al. 2011. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *ACM SIGMOD*. 37–48.
- [3] Sebastian Breß et al. 2017. Generating Custom Code for Efficient Query Execution on Heterogeneous Processors. *CoRR* abs/1709.00700 (2017).
- [4] Max Heimel, Michael Saecker, Holger Pirk, et al. 2013. Hardware-Oblivious Parallelism for In-Memory Column-Stores. *PVLDB* 6, 9 (2013), 709–720.
- [5] Intel. [n. d.]. *Intel C++ Intrinsic Reference*. Retrieved September 30, 2017 from <https://software.intel.com/sites/default/files/a6/22/18072-347603.pdf>
- [6] Saurabh Jha, Bingsheng He, et al. 2015. Improving main memory hash joins on intel xeon phi processors: An experimental approach. *PVLDB*, 642–653.
- [7] Holger Pirk, Oscar Moll, Matei Zaharia, et al. 2016. Voodoo-a vector algebra for portable database performance on modern hardware. *PVLDB*, 1707–1718.
- [8] Orestis Polychroniou, Arun Raghavan, and Kenneth A Ross. 2015. Rethinking SIMD vectorization for in-memory databases. In *ACM SIGMOD*. 1493–1508.
- [9] Stefan Richter, Victor Alvarez, et al. 2015. A Seven-dimensional Analysis of Hashing Methods and Its Implications on Query Processing. *PVLDB*, 96–107.
- [10] Thomas Willhalm et al. 2009. SIMD-scan: Ultra Fast In-memory Table Scan Using On-chip Vector Processing Units. *PVLDB* 2, 1, 385–394.
- [11] Thomas Willhalm, Ismail Oukid, Ingo Müller, and Franz Faerber. 2013. Vectorized Database Column Scans with Complex Predicates. In *ADMS*. 1–12.
- [12] Yang Ye, Kenneth A. Ross, and Norases Vesdapunt. 2011. Scalable Aggregation on Multicore Processors. In *ACM DaMoN*. 1–9.
- [13] Jingren Zhou and Kenneth Ross. 2002. Implementing database operations using SIMD instructions. In *ACM SIGMOD*. 145–156.

Histogram Domain Ordering for Path Selectivity Estimation

Nikolay Yakovets, Li Wang,
George Fletcher
TU Eindhoven, Netherlands
{hush@,l.wang.3@student.,g.h.l.
fletcher@}tue.nl

Craig Taverner
Neo4j, Sweden
craig.taverner@neo4j.com

Alexandra Poulouvasilis
Birkbeck, University of London, UK
ap@dcs.bbkc.ac.uk

ABSTRACT

We aim to improve the accuracy of path selectivity estimation in graph databases by intelligently ordering the domain of a histogram used for estimation. This problem has not, to our knowledge, received adequate attention in the research community. We present a novel framework for the systematic study of path ordering strategies in histogram construction and use. In this framework, we introduce new ordering strategies which we experimentally demonstrate lead to significant improvement of the accuracy of path selectivity estimation over current strategies. These positive results highlight the fundamental role that domain ordering plays in the design of effective histograms for efficient and scalable graph query processing.

1 INTRODUCTION

Analytics on graph-structured data is increasingly important in a variety of domains, e.g., role discovery in social networks, impact analysis in citation networks, functional analysis of biological networks, and querying knowledge graphs. Querying in graph query languages such as openCypher and PGQL is at the heart of these analytics tasks [1, 3, 11]. However, current graph database systems have difficulty in scaling query processing as the size and complexity of graph data collections continue to grow [4, 9].

Towards addressing this challenge, a crucial step in scalability of graph databases is the generation of effective query execution plans. Query optimizers rely on accurate data statistics for cardinality estimation during plan generation. Histograms are among the most widely used data structure for maintaining statistics for cardinality estimation, in particular for relational database systems [5]. However, there has been relatively little work on histograms for graph queries, even for the most basic graph query building block, namely, path queries [6–8, 10].

Our contributions. In this paper, we give an overview of findings in our ongoing investigations into histograms for path selectivity estimation [12]. We focus in particular on ordering strategies for path queries, i.e., how to order the domain over which histograms are built, with the goal of minimizing the variance within histogram buckets (and thereby improving estimation accuracy). We present a novel framework for systematically introducing ordering strategies, showing experimentally that the choice of domain ordering is a fundamental aspect of effective histograms. We introduce new ordering strategies which we demonstrate lead to significant improvement on the accuracy of obtained estimates, over current ordering approaches.

State of the art. The study and efficacy of histogram-based cardinality estimation are well-established [5], e.g., for path and twig query optimization in XML databases [2, 13]. Several studies have also considered path selectivity estimation on graph data

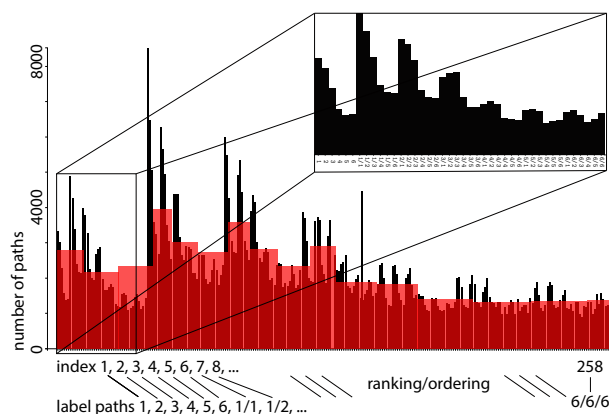


Figure 1: Visualization of a data distribution (black) and an equi-width histogram (red) of Moreno Health dataset with $k = 3$.

with cycles (i.e., beyond trees and DAGs) [6–8, 10]. These works, however, have not investigated histograms or the impact of path ordering on estimation quality. To the best of our knowledge, we present here the first systematic study of this basic aspect of histogram construction and use in graph data management.

2 HISTOGRAMS ON LABEL-PATHS

We investigate selectivity estimation of path queries on graphs. A graph G is composed of a finite set of vertices V , a set of edge labels L , and a set of directed labeled edges $E \subseteq V \times L \times V$. A k -label path is a sequence $\ell = l_1 / \dots / l_k$, where $l_i \in L$, for all $1 \leq i \leq k$. We say $k = |\ell|$ is the length of ℓ . Viewing ℓ as a path query, the evaluation of ℓ on G returns the set $\ell(G)$ consisting of all pairs of vertices (v_s, v_t) in G such that there exist vertices $v_0, v_1, \dots, v_k \in V$ where $v_s = v_0$, $v_t = v_k$, and for $0 < i \leq k$, $(v_{i-1}, l_i, v_i) \in E$. The total number of such pairs, i.e., the cardinality of $\ell(G)$, is called the selectivity of ℓ on G , which we denote by $f(\ell)$.

Let \mathcal{L}^k be the set of all label paths over L with length up to k .¹ An ordering of \mathcal{L}^k is a bijection from \mathcal{L}^k to integer set $[0, |\mathcal{L}^k|)$. Once we establish an ordering on a label path set, a label path can be represented by its positional index in the ordering. For each label path ℓ , let $index(\ell)$ denote the index of ℓ in the ordering.

A histogram is a mechanism used to provide the approximation of frequency for a given value (point query) or value range (range query) without storing or accessing the complete original data distribution. More precisely, given an attribute X , a histogram on this attribute is constructed by partitioning the data distribution of X into $\beta \geq 1$ mutually disjoint subsets called buckets and storing the statistics information and bucket boundaries for each bucket. In this work, attribute X , also called the domain of the

¹We will let \mathcal{L} denote a label path set regardless of k when this does not cause ambiguity.

histogram, is an ordered label path sequence produced by an ordering of \mathcal{L}^k . Then, given label path ℓ and its index $index(\ell)$, such a *label-path histogram* is used to compute an estimate $e(\ell)$ of the selectivity $f(\ell)$. An example of a label-path histogram is shown in Figure 1.

3 ORDERING FRAMEWORK

The purpose of histogram domain reordering is to ensure that label paths with similar cardinality are located close to each other, such that they can be allocated in the same bucket. This leads to lower variance, lower error rates, and overall better quality.

An intuitive and ideal way is to arrange the data distribution such that when $index(\ell)$ increases, $f(\ell)$ monotonically increases or decreases. The most straightforward, yet not feasible, approach is to sort the label paths by their selectivity and assign the *index* of each label path as its position in this sequence. This idea is not practical, however, as it requires extra memory to store $|\mathcal{L}|$ *index* values. The exact amount of memory can also be used to store the cardinality for each label path, such that instead of returning an estimation of selectivity, we can obtain the precise selectivity. We call such ordering an *ideal ordering*. Despite an ideal ordering being prohibitive, we can still construct an approximately monotonic sequence based on the awareness of precise cardinalities of a subset of \mathcal{L} .

For example, by looking at Figure 1, one can observe that the label 1 has the highest cardinality among all length-1 label paths while label 5 has the lowest. Similar trend repeats in the other 6-member groups with the same prefix $\{1/1, 1/2, \dots, 1/6\}$, $\{2/1, 2/2, \dots, 2/6\}$, and so on. Hence, we can assume that the label path that is composed of label paths with high cardinalities should also have high cardinality.

3.1 Concepts

We define a *base label set* as a $B \subseteq \mathcal{L}$ such that every label path in \mathcal{L} can be decomposed into pieces which are all in B .² Then, a *splitting rule* defines how to decompose a label path. For example, \mathcal{L}^6 on Moreno Health dataset is $\{1, 2, 3, 4, 5, 6, 1/1, \dots, 6/6/6/6/6/6\}$, if we choose B to be \mathcal{L}^2 , with a *greedy splitting rule* which at each split step always cuts a piece in B as long as possible. For example, label path “4/4/3/3/6” is decomposed into “4/4”, “3/3” and “6”.

An ordering method can be described by the following three components. First, we need a base label set B . Second, we define (*un*)*ranking function* over the base label set that gives a rank for each base label and vice-versa. It is a bijection which maps between edge label set B and integer set $[1, |B|]$. Finally, we construct an *ordering rule* which is combined with a ranking rule to eventually determine the index of a label path (sequence of base labels) in \mathcal{L}^k . It is a bijection that maps between label path set \mathcal{L} and integer set $[0, |\mathcal{L}^k|]$. A complete ordering method, therefore, is seen as the combination of a ranking rule and an ordering rule on a given dataset. We refer to an ordering method that is composed of ranking rule A and ordering rule B as B - A ordering.

We define two ranking rules in our study. *Alphabetical ranking* assigns ranks based on the alphabetical order of base labels. *Cardinality ranking* is ranking based on the cardinality of base labels, which places a base label with lower cardinality in front of the label with higher cardinality, i.e., $l_1 <^{card} l_2 \iff f(l_1) < f(l_2)$

²Naturally, $L \subseteq B$, otherwise there might exist label paths which cannot be decomposed into label paths in B .

In this work, we focus on the approach that takes the edge label set as the base label set, i.e., $B = L$. We define two bijections: *alph* and *card*. Let $alph(l)$ and $card(l)$ denote the index of edge label l , which will be referred to as the *rank* of l , in the set L totally ordered by alphabetical order and cardinality, respectively.

3.2 Numerical and Lexicographical Orderings

In numerical ordering, each rank is an integer, and a composition of ranks produces a number in $|B|$ -based numeral system. For example, to compare two label paths $\ell_1 = l_1^1/l_2^1/\dots/l_m^1$ and $\ell_2 = l_1^2/l_2^2/\dots/l_n^2$, if one is shorter than the other then it has a lower ranking (rule (1) below), otherwise the two paths’ labels are compared pairwise until a pair of different values is found at position i (rule (2) below):

$$\ell_1 < \ell_2 \iff \begin{cases} |\ell_1| < |\ell_2| & |\ell_1| \neq |\ell_2| \quad (1) \\ \bigwedge_{j=1}^{i-1} (l_j^1 = l_j^2) \wedge (l_i^1 < l_i^2) & |\ell_1| = |\ell_2| \quad (2) \end{cases}$$

Lexicographical ordering is the same as the ordering rule used in dictionaries; it is similar to numerical ordering with the following difference. Instead of comparing lengths of two label paths first, we append $k - |\ell|$ *blank* symbols (i.e., special symbols for which $\forall l \in L, rank(blank) > rank(l)$) to every ℓ to form a length- k sequence. We can then apply Formula 2 to compare the resulting label paths. The time complexity of both ranking and unranking functions for numerical and lexicographical orderings is $O(k)$.

3.3 Sum-based Ordering

Given label path ℓ , the idea of *sum-based* ordering is to use the sum of ranks of all base labels in ℓ to approximate the cardinality of ℓ . While being conceptually simple, the implementation of this ordering method is not trivial. First, given a path label ℓ of length k , ℓ is split into base labels and an integer rank is computed for each of the base labels to obtain a k -length integer *permutation*. Then, the integer permutation of ℓ is mapped to $index(\ell)$ by performing a *three-stage partitioning* of a histogram domain as follows.

The first stage partitions the histogram domain according to the length of the integer permutations, with shorter lengths being assigned partitions with lower indexes in the domain. Then, the size of each of the stage-one partitions can be computed by the following formula (where n is the length of the permutation):

$$sum_n = |L|^n$$

The second stage performs further division of stage-one partitions by grouping all m -length permutations by their *summed ranks*. Those permutations with lower summed rank will have a lower index within a stage-one partition:

$$sr_m = \sum_{i=0}^{m-1} rank(l_i)$$

To compute the boundaries of each of the stage-two partitions, we need to determine how many label paths are in the group with a certain m and sr_m . This question is the same as how many ways there are to distribute sr_m indistinguishable balls over m distinguishable bins of finite capacity $|L|$ with at least one ball in each bin. From combinatorics’ *inclusion-exclusion principle* we have:

$$dist(sr_m, m, L) = \sum_{j \geq 0} (-1)^j \binom{m}{j} \binom{sr_m - j \cdot |L| - 1}{m-1} \quad (3)$$

The third stage explores combinations inside each of the stage-two partitions marked by length m and summed rank sr_m . These combinations are all *integer partitions* of sr_m into exactly m parts, where each part is less than $|L|$. Let integers v, b represent sr_m and $|L|$ respectively. A general formula for integer partition $ip(v, b, m)$ is as follows:

$$ip(v, m, b) = \bigcup_{i=0}^{\lfloor v/b \rfloor} ip(v - i \cdot b, m - 1, b - 1, \underbrace{b, \dots, b}_{i \text{ bs}}) \quad (4)$$

Based on Formula 4, we present a partitioning algorithm which outputs all combinations in the desired cardinality-based order and has time complexity is $O(\log(|L|)^k)$ [12].

Finally, to compute the boundaries of each of the stage-three partitions, we need to determine how many permutations we skip when we skip a stage-three partition. This is equivalent to identifying how many permutations can be generated by a certain combination in which there might be duplicates. Let C denote the combination, d_i denote the number of times an integer i occurs in C , then the number of permutations is given by the following formula:

$$nop(C) = \frac{|C|!}{\prod_{i \in \{0, \dots, |L|-1\}} d_i!} \quad (5)$$

Algorithm 1 finds the combination to which the target permutation belongs and has time complexity of $O(k^2)$.

Algorithm 1 Unranking permutation of combination

```

1: procedure UNRANKING_PERMUTATION(index, C)
2:   if  $i < 0 \vee i \geq nop(C)$  then
3:     return null
4:   end if
5:   if  $|C| = 1$  then
6:     return  $[C[0]]$ 
7:   end if
8:    $i \leftarrow 0$ 
9:   while  $i < |C|$  do
10:     $S \leftarrow C \setminus [C[i]]$  ▷ subset of  $C$ 
11:    if  $index \geq nop(S)$  then
12:       $index \leftarrow index - nop(S)$ 
13:       $i \leftarrow i + count(C, C[i])$ 
14:    continue
15:    else
16:       $sub \leftarrow unranking\_permutation(index, S)$ 
17:       $sub.add(0, C[i])$ 
18:      return sub
19:    end if
20:  end while
21: end procedure

```

Algorithm 2 illustrates the complete version of unranking permutation in sum-based order and has time complexity of $O(\log(|L|)^k)$.

3.4 Ordering Example

We illustrate the proposed ordering methods with examples on an artificial dataset which has 3 unique edge labels and its label paths set with k up to 2. Consider the cardinalities 20, 100, and 80 for edge labels “1”, “2”, and “3”, respectively. Then, for the summed ranks shown in Table 1, label paths arranged in the corresponding

Algorithm 2 Unranking in sum-based order

```

1: procedure UNRANKING_IN_SUMBASED(index, L, k) ▷ index,
   edge label set, k
2:   if  $index < 0 \vee index > |\mathcal{L}^k|$  then
3:     return null
4:   end if
5:   for  $len \in 1, \dots, k$  do
6:     if  $index \geq |L|^{len}$  then
7:        $index \leftarrow index - |L|^{len}$ 
8:     continue
9:   end if
10:  for  $sum \in len, \dots, len * |L|$  do
11:    if  $index \geq dist(sum, len, |L|)$  then
12:       $index \leftarrow index - dist(sum, len, |L|)$ 
13:    continue
14:  end if
15:   $P \leftarrow ip(sum, len, |L|)$ 
16:  for  $p \in P$  do
17:    if  $index \geq nop(p)$  then
18:       $index \leftarrow index - nop(p)$ 
19:    continue
20:  end if
21:   $p' \leftarrow \{i - 1 | i \in p\}$ 
22:   $sort(p')$ 
23:  return  $unranking\_permutation(index, p')$ 
24: end for
25: end for
26: end procedure

```

Label Path	1	2	3	1,1	1,2	1,3	2,1	2,2	2,3	3,1	3,2	3,3
Summed Ranks	1	3	2	2	4	3	4	6	5	3	5	4

Table 1: Summed ranks

Index \ O	0	1	2	3	4	5	6	7	8	9	10	11
<i>num-alpha</i>	1	2	3	1,1	1,2	1,3	2,1	2,2	2,3	3,1	3,2	3,3
<i>num-card</i>	1	3	2	1,1	1,3	1,2	3,1	3,3	3,2	2,1	2,3	2,2
<i>lex-alpha</i>	1	1,1	1,2	1,3	2	2,1	2,2	2,3	3	3,1	3,2	3,3
<i>lex-card</i>	1	1,1	1,3	1,2	3	3,1	3,3	3,2	2	2,1	2,3	2,2
<i>sum-based</i>	1	3	2	1,1	1,3	3,1	3,3	1,2	2,1	3,2	2,3	2,2

Table 2: Ordered label paths according to different ordering methods O

orderings are shown in Table 2. Respectively, numerical ordering associated with alphabetical ranking, numerical ordering with cardinality ranking, lexicographical ordering with alphabetical ranking, lexicographical ordering with cardinality ranking, sum-based ordering with cardinality ranking are referred to as *num-alpha*, *num-card*, *lex-alpha*, *lex-card* and *sum-based*.

4 EXPERIMENTAL STUDY

We implemented a k -path histogram construction and path selectivity estimation in Java. All experiments are conducted on an Ubuntu 16.04 machine equipped with an Intel i5 CPU with 4GB of RAM. We use the datasets shown in Table 3. The goal of our experiments is two-fold. First, we verify the impact of different domain ordering techniques on the estimation time. Second, we showcase the gains in estimation accuracy which can be obtained by using sum-based histogram domain ordering.

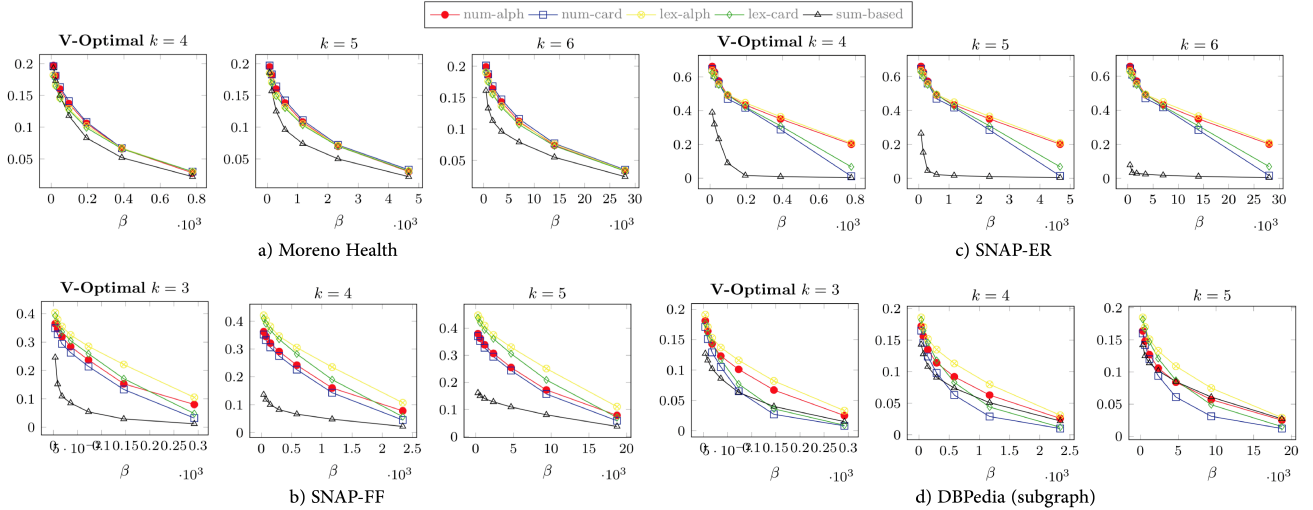


Figure 2: Mean error rate of estimation for different domain ordering techniques on V-Optimal k -path histogram

Dataset	#Edge Labels	#Vertices	#Edges	Real world data
Moreno health ³	6	2539	12969	yes
DBpedia (subgraph) ⁴	8	37374	209068	yes
SNAP-ER ⁵	6	12333	147996	no
SNAP-FF	8	50000	132673	no

Table 3: Datasets

β	Average Estimation Running Time (in ms)				
	num-alpha	num-card	lex-alpha	lex-card	sum-based
27993	9.98	8.62	9.65	8.7	11.02
13996	7.69	7.23	7.79	7.3	9.39
6998	7.36	6.8	7.07	6.93	8.55
3499	6.4	6.52	5.97	6.31	7.42
1749	5.71	5.76	5.76	5.21	6.64
874	5.8	5.06	5.78	5.18	6.1
437	5.19	4.58	4.52	4.29	6.13

Table 4: Average estimation execution time in V-optimal histogram with different ordering methods (in ms)

Performance. We study the execution time of estimation associated with different ordering methods as follows. For $k = 6$, five V-optimal histograms are built, each of which is associated with an ordering method: *num-alpha*, *num-card*, *lex-alpha*, *lex-card*, and *sum-based*. The total number of label paths is 55996. We run 7 experiments by varying the number of buckets (β) in each histogram. All experiments are executed 100 times and the average estimation time is taken. The results (Table 4) demonstrate that sum-based ordering is approximately 20% slower in estimation than native ordering methods. This is explained by the higher complexity of the sum-based (un)ranking function.

Accuracy. We measure the average estimation accuracy by constructing a V-optimal histogram for each ordering method for varying k and β (Figure 2). We use the following $err(\ell)$ metric to measure the error of an estimation:

$$err(\ell) = \begin{cases} 0 & \text{if } e(\ell) = f(\ell) \\ \frac{e(\ell) - f(\ell)}{\max(e(\ell), f(\ell))} & \text{else} \end{cases} \quad (6)$$

³http://konect.uni-koblenz.de/networks/moreno_health

⁴<http://wiki.dbpedia.org>

⁵<https://snap.stanford.edu/snappy/>

We observe that, for the synthetic datasets, sum-based ordering provides accuracy which is far superior to other ordering methods, especially, for histograms with a low number of buckets. For the real-life datasets, the performance difference is not as significant, but still observable. This can be explained by the presence of edge-label cardinality correlations in real-life data.

5 CONCLUDING REMARKS

We have reported on initial findings in our ongoing study of domain ordering for improving histogram-based path selectivity estimation. Experimental study has demonstrated the promise of our framework, which facilitates the further systematic study of effective histogram design for graph databases. A primary future research direction is to expand the framework with additional ordering strategies, e.g., those built over richer base sets such as \mathcal{L}^2 , towards capturing correlations between label paths.

REFERENCES

- [1] 2017. openCypher. (2017). <https://www.opencypher.org/>
- [2] A. Aboulmaga, A. Alameldeen, and J. Naughton. 2001. Estimating the selectivity of XML path expressions for internet scale applications. In *VLDB*. 591–600.
- [3] Renzo Angles et al. 2018. G-CORE: A core for future graph query languages. In *SIGMOD 2018*. to appear.
- [4] G. Bagan, A. Bonifati, R. Ciucanu, G. Fletcher, A. Lemay, and N. Advokaat. 2017. gMark: Schema-driven generation of graphs and queries. *IEEE Trans. Knowl. Data Eng.* 29, 4 (2017), 856–869.
- [5] G. Cormode et al. 2012. Synopses for massive data: samples, histograms, wavelets, sketches. *Found. Trends Data*. 4, 1-3 (2012), 1–294.
- [6] George H. L. Fletcher, Jeroen Peters, and Alexandra Poulouvasilis. 2016. Efficient regular path query evaluation using path indexes. In *EDBT 2016*. 636–639.
- [7] T. Neumann and G. Moerkotte. 2011. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *ICDE 2011*. 984–994.
- [8] Yun Peng, Byron Choi, and Jianliang Xu. 2011. Selectivity estimation of twig queries on cyclic graphs. In *ICDE 2011*. 960–971.
- [9] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2017. The ubiquity of large graphs and surprising challenges of graph processing. *PVLDB* 11 (2017), 420 – 431. Issue 4.
- [10] Silke Trüßl and Ulf Leser. 2010. Estimating result size and execution times for graph queries. In *GraphQ 2010*. 11–20.
- [11] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. 2016. PGQL: a property graph query language. In *GRADES 2016*.
- [12] Li Wang. 2017. *On histograms for path selectivity estimation in graph data*. Master’s thesis. Eindhoven University of Technology.
- [13] Yuqing Wu, Jignesh M. Patel, and H. V. Jagadish. 2003. Using histograms to estimate answer sizes for XML queries. *Inf. Syst.* 28, 1-2 (2003), 33–59.

Nomadic Datacenters at the Network Edge: Data Management Challenges for the Cloud with Mobile Infrastructure

[Vision Paper]

Faisal Nawab
UC Santa Cruz
Santa Cruz, CA
fnawab@ucsc.edu

Divyakant Agrawal, Amr El Abbadi
UC Santa Barbara
Santa Barbara, CA
[agrawal,amr]@cs.ucsb.edu

ABSTRACT

The continuing growth and success of many edge technologies such as the Internet of Things (IoT), wearables and Virtual and Augmented Reality (VR/AR) relies on providing a high-performance and low-latency computing infrastructure. In this paper, we envision extending edge computing with mobile, moving, and possibly flying edge datacenters, that we call *nomadic datacenters* to improve the performance and capacity of the edge infrastructure. In particular, we study how the introduction of nomadic datacenters will affect data management systems and find that novel challenges and opportunities need to be addressed. We present some of these challenges and opportunities in addition to an outline of how they can be tackled by future data management systems.

1 INTRODUCTION

Emerging classes of computing technologies are promising to transform our lives, change how we interact with each other and with the world. These include Internet of Things (IoT), wearables, and Virtual and Augmented Reality (VR/AR). IoT enables harnessing the multitude of sensor data via applications ranging from smart farming to autonomous cars. Wearable technology enables personalized applications such as activity tracking and health monitoring. With VR/AR, we will be immersed in designed experiences that will touch every facet of our lives. Common among these transformative technologies are the utilization of sophisticated edge devices and the demand for high-throughput and/or low-latency. We will use the term *edge technologies* to denote IoT, wearable, and VR/AR technologies in light of their common characteristics.

To realize the potential of edge technologies, it is necessary to provide the hardware and software infrastructures that support the high-throughput and low-latency demands for application processing. To this end, many efforts have advocated for the use of edge computing technology to provide more compute and storage power [4, 13]. Edge computing enhances cloud computing by introducing *edge datacenters* that are closer to users and that consist of a few to hundreds of machines. With edge computing, data can be processed at the edge, saving the communication latency to the datacenter that can take up to 100s of milliseconds [13]. Additionally, processing at edge datacenters saves the monetary bandwidth costs of cloud-edge communication.

Realizing the benefits of edge datacenters has been limited by the static nature of edge datacenter deployment. Typically, edge datacenters are rigidly stuck in fixed locations. This introduces two limitations:

- *A static deployment cannot follow the hot spots:* It is difficult for a static deployment to adapt to a dynamic, mobile environment, where the current location of the highest data traffic is continuously, and often unpredictably changing. For example, consider building an edge infrastructure to support a taxi transportation organization. Taxi cabs connect to edge datacenters that are placed in various locations around the city. However, the location of taxi cabs depends on many volatile aspects such as traffic and passengers. This makes it very difficult to decide where to place the edge datacenters and how to provision resources around the city.
- *A static deployment cannot be recovered swiftly:* In the event of a natural disaster and power outages, a large area may be affected. The edge datacenters across these large areas may be inaccessible or even permanently damaged. To recover from such catastrophic failures that damage infrastructure, replacing the infrastructure is necessary. However, it may take days to replace and deploy the new infrastructure. This is especially devastating when the edge infrastructure is needed to aid in responding to natural disasters.

In this paper, we propose extending edge computing technology with dynamic, mobile edge datacenters, and call them **nomadic datacenters**. Nomadic datacenters denote small, portable edge datacenters, that can be relocated swiftly by large vehicles (e.g., trucks) and air crafts (e.g., helicopters). Nomadic datacenters can overcome the two limitations of static edge datacenters that we outlined in the previous paragraph. A nomadic datacenter *can* follow the hot spot. In the taxi organisation example, nomadic datacenters can be continuously moving to maximize the utility of resources. Also, nomadic datacenters can replace a damaged infrastructure swiftly in cases of natural disasters and power outages. In fact, nomadic datacenters can be thought of as an aid to first responders that may need the edge computing resources to collect and process data in addition to providing connectivity during relief and rescue operations.

The potential feasibility of the concept of nomadic datacenters is due to recent advances in datacenter and communication technology. Edge datacenter technology has been continuously improving during the past decade, resulting in small, containerized datacenters, also called *micro datacenters*. A micro datacenter may contain as little as a few servers on a single rack with built-in cooling, power supply/backup, and fire suppression systems.

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

Already, these micro datacenters are being leveraged for their portable nature that allows deploying them in remote areas such as shallow- and deep-water oil rigs [1]. Our proposal is to leverage this portability to react to dynamic, mobile edge applications and actively follow hotspots in addition to replacing damaged edge infrastructure.

The other technology that is enabling nomadic datacenters is 5th generation mobile networks (5G). 5G is 4G's successor telecommunication standard. It aims to provide lower latency, higher communication throughput, and low power transmission. More importantly, 5G is geared towards supporting emerging edge applications, such as ones based on IoT and wearables. This is envisioned by providing support of device-to-device and mobile broadband communication. This is significant for the realization of nomadic datacenters that will rely on wireless communication to connect to edge users/devices and the cloud. 5G will allow larger-scale communication between a nomadic datacenter and edge users and devices through device-to-device and mobile broadband communication. Also, nomadic datacenters would need a large capacity wireless link to connect to the backbone in the cloud. 5G's larger capacity will ameliorate the capacity limits of wireless telecommunication technology compared to wired and optical fiber communication. Also, nomadic datacenters would typically be powered for long (or all) durations on batteries. Thus, low power consumption based on 5G is an important feature.

The concept of a nomadic datacenter is not new, as many have suggested the idea for its practical uses in many applications [12]. However, it was not realized because the relevant communication and datacenter technologies were not ready. With the advances of micro datacenters and 5G telecommunication, nomadic datacenters are positioned to be a reality now more than ever.

In this paper, we envision the last piece in the puzzle of making nomadic datacenters a feasible technology—we propose the study of data management systems for nomadic datacenters. In addition to the hardware and communication infrastructures (i.e., micro datacenters and 5G) to realize nomadic datacenters, data management technology must be revisited to tackle the unique challenges and exploit the opportunities of the new edge architecture. We present a system model that encompasses several scenarios of how nomadic datacenters will be realized. Then, our study of the system model reveals a set of novel data management challenges and opportunities. We present these challenges, opportunities and a roadmap to tackle them.

2 A SYSTEM MODEL FOR NOMADIC DATACENTERS

In this section, we present our vision of emerging system models of nomadic datacenters. The development of these system models is important to guide the design decisions and identification of the salient properties of the new technology. We begin by describing the base architecture of nomadic datacenters and its interaction with the cloud, users, and other edge datacenters. Then, we discuss how the model can be adapted to various properties of nomadic datacenter deployments that have varying sizes and mobility characteristics.

The base system model consists of three tiers (shown in Figure 1):

- (1) *Cloud tier*: This denotes the cloud resources at traditional, large datacenters.

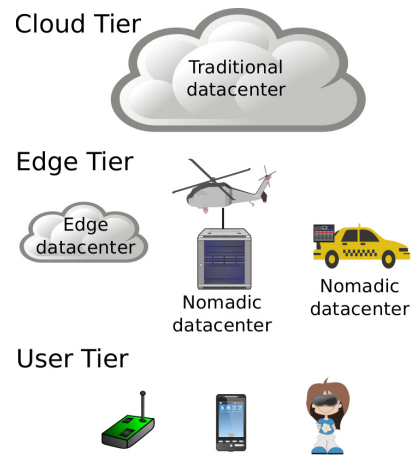


Figure 1: The system model with nomadic datacenters.

- (2) *Edge tier*: This denotes the resources at edge and nomadic datacenters. Edge datacenters communicate with the cloud tier through high-bandwidth links, such as fiber optics, and nomadic datacenters communicate with the cloud tier via wireless telecommunication (e.g., 5G). However, nomadic datacenters are also capable of communicating with other edge datacenters through wireless links. This will enable coordination between nodes in the edge tier and also enables relaying communication between nomadic datacenters and the cloud tier through intermediate edge datacenters.
- (3) *User tier*: This denotes the users and devices generating data and making requests. Nodes in the user tier, that we will call user nodes, communicate with the application through the edge tier, if an edge or nomadic datacenter is nearby. Typically, this communication would be through wireless links, using Wi-Fi technology. If no edge or nomadic datacenter is nearby, then the user nodes communicate directly with the cloud tier.

Nomadic datacenters will vary in size to adapt to various application and environment requirements. We abstract the different sizes of nomadic datacenters to fall in one of three sizes: (1) Light: This represents nomadic datacenters that contain a single machine and minimal datacenter capabilities for cooling and security. This is ideal in cases where the nomadic datacenter should be carried by small vehicles, such as drones. (2) Medium: This represents nomadic datacenters that contain a few machines and is ideal to be carried by small vehicles such as taxi cabs. (3) Heavy: This represents nomadic datacenters that resemble current micro datacenters that contain 4 or more machines with various datacenter capabilities. This is ideal for cases where a truck or an aircraft carries the datacenter. The size of the nomadic datacenter influences its mobility characteristics. For example, light nomadic datacenters can be deployed for high mobility scenarios with constant movement and relatively smaller vehicles (e.g., drones). Medium ones can be deployed on medium-sized vehicles, and thus can be used for mobility cases in urban settings. Heavy nomadic datacenters has relatively restricted mobility and are ideal in cases where mobility is in reaction to a non-frequent event.

Given this view of the system model of nomadic datacenters, we discuss some data management issues in the context of nomadic datacenters.

	Centralized Infrastructure	Extended Infrastructure
Static Resources	Client-Server (Cloud Computing)	Edge Computing
Mobile Resources	MDMS	Nomadic Datacenters

Table 1: The high-level differences in the system model of Nomadic datacenter in comparison to other system models.

3 DATA MANAGEMENT FOR NOMADIC DATA CENTERS

In the nomadic datacenter architecture, is there a need for innovation in data management systems to support the new environment or do existing systems suffice? This section answers this question by discussing how the nomadic datacenter architecture is positioned in relation to early work on mobile databases (Section 3.1). Then, we show how the unique properties of nomadic datacenters require innovation in data management systems in Section 3.2.

3.1 Nomadic Datacenters in the Space of Mobile Data Management

Building data management systems for mobile environments has been studied extensively for a number of decades [3, 6, 8, 9]. In these early works, that we will denote as *mobile data management systems* (MDMS), users and data copies are mobile and may use wireless communication. These properties are similar to the properties brought forth in the nomadic datacenters system model (Section 2).

So, can we just use MDMS [3, 6, 8, 9] to build solutions for nomadic datacenters? Our study of MDMS systems have revealed that they are an excellent starting point and basis for data management solutions for nomadic datacenters. MDMS outlines many solutions to tackle challenges that also arise in nomadic datacenters such as resources asymmetry, mobility, caching, and energy efficiency. Many of these techniques can—and should—be adopted by data management systems for nomadic datacenters.

However, there is a key difference in the nomadic datacenters architecture compared to early MDMS architectures. In MDMS, there is a “core” database and mobile users. This makes MDMS consider a centralized infrastructure with mobile resources. For nomadic datacenters, the users are mobile like MDMS, but the infrastructure is different. The infrastructure in nomadic datacenters is extended beyond a centralized location to mobile edge nodes (*i.e.*, nomadic and edge datacenters).

Table 1 summarizes the high-level position of nomadic datacenters compared to relevant architectures. An interesting relation that can be observed from the table is that *a nomadic datacenters architecture to MDMS is what edge computing is to cloud computing*. Therefore, the data management challenges of edge computing in comparison to cloud computing will likely exist for nomadic datacenters in comparison to MDMS. The nature of these challenges stems from the fact that edge resources are more powerful than client machines, and thus there is an opportunity to leverage them more aggressively for data management tasks like caching [5], and offloading [11, 14]. This makes the caching and offloading results for edge computing [5, 11, 14] different from ones for MDMS [3, 6, 8, 9].

Although transforming edge solutions to adapt to the nomadic datacenter architecture is important and may entail interesting designs, we are interested more in the novel challenges and opportunities that are unique to nomadic datacenter architecture. The next section introduces such challenges and opportunities.

3.2 New Challenges and Opportunities

We postulate that the nomadic datacenter architecture has fundamental differences compared to previous architectures and combinations of previous architectures (*e.g.*, MDMS with edge computing). These differences require innovative solutions to tackle the challenges and benefit from the opportunities of nomadic datacenters. In this section, we present the imminent challenges and opportunities in this space.

3.2.1 The wireless link: a bottleneck and an opportunity. Nomadic datacenters communication with users, other edge and nomadic datacenters, and the cloud tier through wireless links. Although emerging technology such as 5G will ameliorate this bandwidth limitation of wireless links, they are still limited compared to wired infrastructures. In addition, communication between nomadic datacenters and users and other edge datacenters will likely still rely on Wi-Fi or similar technologies. Therefore, *communication bandwidth will be an extremely costly resource for nomadic datacenters*. This is not the case for other architectures. (Unlike MDMS, a nomadic datacenter manages the data of a large number of users and thus require a significantly larger bandwidth than a single MDMS client.) The bandwidth cost significantly affects the design trade-offs space for data management tasks as we outline in the next example.

A case study on coordination. An example of a data management task that will be affected by the high bandwidth cost is coordination. Coordination is necessary to maintain consistency across different copies of data via protocols such as Two-Phase Commit [7], Paxos [10], and others. Coordination is communication-intensive, where each request is typically coordinated among nodes that hold copies of the accessed data. Take for example a scenario of two nomadic datacenters, *A* and *B*, that are located to be close to a large event that is anticipated to generate a lot of traffic. Both nomadic datacenters, *A* and *B*, are providing access to the same data but half the users are connected to *A* and the other half is connected to *B*. Now, in applications where requests would require coordination (such as OLTP transactions), *A* and *B* must coordinate every request they receive. For example, a request that is received at *A* would create a coordination request from *A* to *B*. This means that, potentially, double the wireless link bandwidth is consumed than necessary. This is exacerbated for cases with more nomadic datacenters and represents a major source of wasted resources and stress to the most limited nomadic datacenter resource—communication bandwidth.

Optimizing the bandwidth of coordination is an open challenge for nomadic datacenters and can be tackled based on familiar approaches in batching and compression. However, there are also opportunities for innovative solutions to this problem that are allowed by the unique architecture of nomadic datacenters. Specifically, since all communication is through the wireless medium, *nodes can eavesdrop on other nodes’ communication*. Consider our earlier coordination scenario where *A* receives a request and then coordinate with *B*. Since *B* can eavesdrop on the communication from the user to *A*, then it already knows about the request, even before *A* initiates coordination for it. This allows efficient alternatives to coordination. For example, *B* can signal

that it has heard about the request and decides a certain action regarding it. This is more efficient in two ways: (1) *A* does not have to communicate with *B*, and (2) the communication from *B* is a control message only and does not need to contain the payload of the request that is likely much bigger than control messages. Another example to reduce coordination overhead is partitioning. However, unlike traditional partitioning where the system controls partitioning the data only, in our scenario, the nomadic datacenters can also partition the users between them. This means that *A* and *B* can judiciously decide which users connect to which nomadic datacenter, and in jointly with a data partitioning strategy between *A* and *B*, the need for coordination between *A* and *B* would be reduced.

To summarize, we foresee innovations in coordination and other data management tasks that tackle the main bottleneck of nomadic datacenters (communication bandwidth) via innovative solutions specific to the nomadic datacenter environment such as eavesdropping and users-data partitioning.

3.2.2 Challenges of a Dynamic Mesh Environment. In many of the scenarios we envision, there is a large number of nomadic datacenters that roam around continuously. For example, consider a deployment on taxi cabs, where each taxi cab contains a nomadic datacenter. This may serve an application for the taxi cabs, but may also be serving urban applications. In this scenario, there are potentially thousands of nomadic datacenters that get connected to each other sporadically. We call this a *dynamic mesh model*, akin to Wireless Mesh Networks (WMNs) [2]. However, unlike WMNs, mobility and relocation are rapid and each node is a nomadic datacenter that performs extensive computation and performs data management tasks on behalf of users, rather than just being a communication hub.

The dynamic mesh model introduces serious challenges to distributed protocols that are necessary to manage replicated or partitioned data across nodes. Such distributed protocols, such as Paxos [10], Two-Phase Commit [7], and others, were designed for a static infrastructure, where the participants in a protocol are known. However, in a dynamic mesh model, the participants and topology configuration changes continuously, *requiring the invocation of expensive membership and leader election protocols continuously*. This motivates the study of data management system designs that assume that membership and leader election is invoked frequently.

A case study on Paxos. For example, consider the Paxos protocol [10]. Paxos performs two tasks: leader election and replication. Replication is performed by a leader to make sure data is persistent. Leader election is only invoked during a suspected failure of the current leader. Therefore, many Paxos designs and variants optimize for the case where leader election is rare. Furthermore, Paxos reconfiguration is invoked in cases where a machine is to be replaced or to migrate the infrastructure. Paxos reconfiguration [10] is very expensive and in traditional deployment this is not problematic because it is extremely rare. Because nomadic datacenters move rapidly and the configuration changes continuously, leader election and reconfiguration are invoked frequently. This invites a redesign of the Paxos protocol to make leader election and reconfiguration more efficient. Such redesigns might be enhanced by exploiting the opportunities enabled by the special characteristics of nomadic datacenters. An example is that leader election and reconfiguration are triggered by mobility in nomadic datacenters, rather than failures as the traditional cases. This can be exploited because the current leader is aware

of the anticipated leader election and/or reconfiguration and can participate in it. Therefore, rather than elaborate complex mechanisms to ensure the correct election of a new leader, a live, mobile leader can simply relinquish leadership to another node unilaterally.

Although we discussed a single case study on Paxos, the challenges of membership and reconfiguration in an extremely mobile environment are applicable to a wide-range of distributed protocols. We believe that opportunities to optimize membership and reconfiguration mechanisms for nomadic datacenters exist akin to the high-level optimization that we discussed for the Paxos case study.

4 CONCLUSION

Nomadic datacenters have the potential of enabling a wide-range of edge applications that rely on emerging edge technologies such as IoT, wearables, and Virtual and Augmented Reality. Nomadic datacenters introduces a dynamic, mobile infrastructure that allocate resources on demand in reaction to changes in workload and failures. This makes it suitable for emerging mobile edge applications. In this paper, we outline our vision of nomadic datacenters, how they are becoming more feasible, how they will be realized, and what are the imminent data management problems that arise with their introduction. We find that the areas that require attention in designing data management systems for nomadic datacenters are: the mobility of the extended infrastructure hierarchy, the wireless nature of communication, and the dynamic, mesh nature of the topology. We outline pathways to solutions to these problems and envision that they will provide a building block towards realizing nomadic datacenters.

ACKNOWLEDGEMENTS

This work is supported by NSF grant CSR 1703560.

REFERENCES

- [1] 2017. How Microsoft is Extending Its Cloud to Chevron's Oil Fields. <http://www.datacenterknowledge.com/microsoft/how-microsoft-extending-its-cloud-chevron-s-oil-fields>. (2017).
- [2] Ian F Akyildiz, Xudong Wang, and Weilin Wang. 2005. Wireless mesh networks: a survey. *Computer networks* 47, 4 (2005), 445–487.
- [3] Daniel Barbara and Hector Garcia-Molina. 1993. *Replicated data management in mobile environments: Anything new under the Sun?* Technical Report. Stanford.
- [4] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. 2012. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. ACM, 13–16.
- [5] Ittay Eyal, Ken Birman, and Robbert van Renesse. 2015. Cache serializability: Reducing inconsistency in edge transactions. In *Distributed Computing Systems (ICDCS), 2015 IEEE 35th International Conference on*. IEEE, 686–695.
- [6] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. 1996. The dangers of replication and a solution. *ACM SIGMOD Record* 25, 2 (1996), 173–182.
- [7] Jim Gray and Andreas Reuter. 1992. *Transaction processing: concepts and techniques*. Elsevier.
- [8] Joanne Holliday, Divyakant Agrawal, and Amr El Abbadi. 2002. Disconnection modes for mobile databases. *Wireless Networks* 8, 4 (2002), 391–402.
- [9] Tomasz Imielinski and BR Badrinath. 1994. Mobile wireless computing: Challenges in data management. *Commun. ACM* 37, 10 (1994), 18–28.
- [10] Leslie Lamport. 1998. The Part-time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (May 1998), 133–169. <https://doi.org/10.1145/279227.279229>
- [11] Yi Lin, Bettina Kemme, Marta Patino-Martinez, and Ricardo Jimenez-Peris. 2007. Enhancing Edge Computing with Database Replication. In *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems (SRDS '07)*. IEEE Computer Society, Washington, DC, USA, 45–54. <http://dl.acm.org/citation.cfm?id=1308172.1308219>
- [12] B. Rimler and N. Rasmussen. 2006. Mobile data center. (April 20 2006). <https://www.google.com/patents/US20060082263> US Patent App. 11/251,587.
- [13] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. 2009. The case for vm-based cloudlets in mobile computing. *IEEE pervasive Computing* 8, 4 (2009), 14–23.
- [14] Hemant Saxena and Kenneth Salem. 2015. EdgeX: Edge Replication for Web Applications. In *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*. IEEE, 1041–1044.

Dynamic Resource Routing using Real-Time Information

Sebastian Schmoll
Ludwig-Maximilians-Universität
Munich, Germany
schmoll@db.s.i.fi.lmu.de

Matthias Schubert
Ludwig-Maximilians-Universität
Munich, Germany
schubert@db.s.i.fi.lmu.de

ABSTRACT

Searching the nearest available resource is an important query with many applications in spatial database systems. Examples are searching free parking spots or charging stations in a road network. Information about the availability is usually approximated as long-term statistics. Recently, online systems and sensor networks become more and more common providing access to real-time information about resource availability. In this paper, we consider searching the next available resource in a road network considering real-time information about all resources in a target area. To make the best use of this information, it is not enough to predict a static result route. Instead we propose to model the problem as a Markov decision process (MDP) and compute a routing policy which allows flexible reaction to newly observed developments. In our experiments, we demonstrate that our new approach is capable to exploit the additional information and outperforms previous approaches built on long-term distributions on a real-world parking data set.

1 INTRODUCTION

In modern location based services, queries often aim at finding a spatial resource of a specific type. In contrast to a point of interest (POI), a spatial resource is not generally available but might be occupied at a certain point in time. Correspondingly, an occupied resource might become available during search time. Examples for spatial resources are parking spots, charging stations, rental vehicles or drop-off locations for rental vehicles. To handle the availability of spatial resources, online systems can provide real-time information about currently available resources. However, there is no guarantee that a currently available resource will be available at the arrival time of our user. One might argue that a reservation service might solve any such problem, but reservation services often add additional complexity which often makes them impractical. Reserved resources might be used without authorization and generally available resources might remain unused due to just-in-case reservations. For example, a parking spot might be occupied even if it was reserved before because the previously parked vehicle was not removed in time. Correspondingly, inner-city parking spots might remain unused due to rich people having a permanent reservation just in case they want to go into town. To conclude, for many applications reservation systems might be unreliable and expensive to enforce. Thus, we focus on the case that a resource is claimed by the first user to arrive.

We investigate the task of finding an available resource spending minimal travel time or any other type of cost. Furthermore, we assume real-time information on all available and occupied resources in a specified target area. To properly exploit the available real-time information, a simple route is not enough. Since

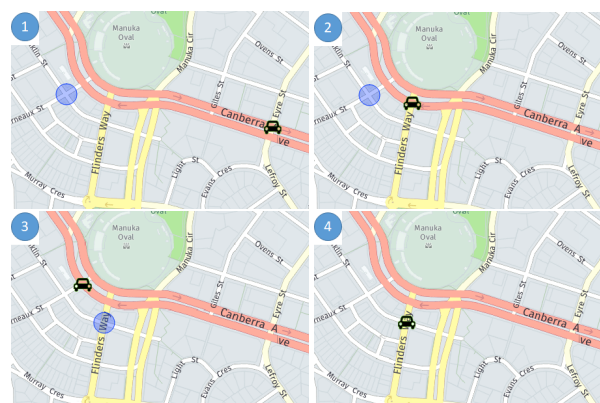


Figure 1: Example for a parking search using our new algorithm. The car symbol denotes the car whose driver is looking for a parking spot. The blue circles denote at least one free available parking slot at the corresponding location.

the availability of resources might be constantly changing, a fixed route cannot include reactions to developments during the search. Thus, we need to compute a policy which indicates the most promising action for each situation during the search. For example, when looking for a parking spot a natural behavior is to take any parking spot getting available along the way. A resource route as proposed in [5] would follow its fixed search order which is based on the information provided at search time. Thus, a resource route would only consider the vacant spot if it is scheduled in the route. In contrast, a policy driven system would recognize the improved situation and propose to take the available parking spot if it is a suitable location.

In this paper, we propose to employ the statistical model of Markov decision processes (MDP) to learn such a policy. A policy is a mapping of any possible situation to the most promising action which an agent should perform to maximize the probability of success. Thus, our proposed solution allows that resources might get occupied or available during the search. The availability of each resource is modeled as a time-continuous Markov chain. Based on these probabilities, we define an MDP to model resource search with real-time information. A drawback of this model is that the state space grows exponentially with the number of acceptable resources. Since in many cases resources are stacked at particular areas, we propose resource aggregation to limit the number of considered states. Additionally, we propose to precompute policies for relevant target areas and store them in a data structure for efficient retrieval of the proposed action. To conclude, the contributions of this paper are as follows:

- A model to compute a search policy for resource search considering real-time information.
- A method for precomputing policies and apply the learned policy for guiding a user.

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

- Experiments on real-world parking data demonstrating the advantage of using real-time information.

The rest of the paper is organized as follows: Section 2 surveys previous works on resource search and MDPs. In Section 3, we give a brief introduction to MDPs and introduce our model for resource search. Section 4 outlines the use of policies for routing services. In our experimental evaluation in Section 5, we demonstrate the advantages of our new approach. Section 6 concludes the paper with a short summary and directions for future work.

2 RELATED WORK

The methods being closest to our approach are [5] and [3]. [5] describes a method for computing a route minimizing the expected search time in a setting where resources can get occupied and available during the search. However, real-time information is only considered to be available at query time. Thus, the resulting resource route cannot react to updates during the search. In [3], the authors compute a policy for resource search. The solution is basically an MDP and the proposed method resembles the basic value iteration algorithm for computation. The major difference to our work is that the method does not consider the availability of real-time information during the search. However, there are already several prototypes and testbeds providing this type of information for urban areas such as the city of Melbourne in our evaluation. Our method employs Markov decision processes (MDP) as described in [7]. Though there is a plethora of more sophisticated solution methods for computing MDPs like [1, 2, 4, 6, 8, 9], we employ the basic value iteration algorithm since the approach in this paper precomputes policies and afterwards queries these policies to guide a user.

3 PROBLEM SETTING

3.1 Markov Decision Processes

A Markov Decision Process consists of a set of states S , sets of actions $\forall s \in S : A(s)$, the probability distribution for each state action pair $P(s'|s, a)$ and a cost (or reward) function $c(s, a)$. It is important to note that $P(s'|s, a)$ indicates that the result of the action (s, a) is uncertain and that any state transition $s \rightarrow s'$ with $P(s'|s, a) > 0$ for any $a \in A(s)$ is potentially possible.

A policy π is a set of state action pairs (s, a) that determines for any state $s \in S$ exactly one action $a \in A(s)$. In our usage of MDPs, we assume terminal states S^* which indicate the targets of the search. Thus, the goal of a policy π is to reach any terminal state $s^* \in S^*$ and a proper policy guarantees that we will reach a terminal state s^* at some point in time. To compare policies, we now define the expected cost $U(s)$ which is also referred to as utility or value function in literature.

In general, $U^\pi(s)$ of policy π represents the expected cost which is aggregated over all possible state sequences starting with state s . We define the optimal policy π^* for any state s as:

$$\pi^*(s) = \operatorname{argmin}_{a \in A(s)} \sum_{s'} P(s'|s, a) \cdot U^*(s')$$

Whereas $U(s)$ can be computed by the following equation also known as Bellman equation:

$$U(s) = \min_{a \in A(s)} c(s, a) + \gamma * \sum_{s'} P(s'|s, a) \cdot U(s')$$

In other words, the minimal expected cost for state s are achieved by taking the action a where the sum of direct costs $c(s, a)$ and the expected future cost are minimal. The parameter γ is used to

weight direct costs against future costs which is often beneficial for the convergence of computation.

3.2 Modelling Resource Search

After giving a brief general introduction to MDPs, we will now turn to modelling resource queries as an MDP problem. In our setting an agent, e.g., a driver or a person moves in a road network. A road network $G = (N, E)$ consists of a set of nodes N and a set of edges $E \subseteq N \times N$. Each edge $e \in E$ yields traversing costs c_e , e.g., travel time or distance. Additionally, we model resource locations as nodes $P \in N$. For each resource $p \in P$, the boolean function $a(p)$ denotes 0 if p is occupied and 1 if p is available.

The state space of our MDP is a concatenation of the agent position $l \in N$ and the value of $a(p)$ for each $p \in P$. Thus, the number of all possible states is $|N| \cdot 2^{|P|}$. All states where $l \in P$ and $a(l) = 1$ are considered as terminal states because the agent arrives at the location of a currently available resource. The actions in our model consists of the possible routing directions at each crossing, e.g., turn left, turn right, go straight or turn.

After defining states and actions, we now have to provide the likelihoods $P(s'|s, a)$. In our setting, the next location of the agent l' is determined by the selected action a . However, the change of the available resources depends on the status changes of the resources P . Thus, we need to compute the likelihood that a resource p changes its current status from occupied to available or vice versa. Given that we assume exact information at the current point of time, we only need to estimate the distribution of a state change for the required time to travel from l to l' . To do so, we rely on the continuous time Markov model being proposed in [5]. The model predicts the likelihood $P(p(t) = 0)$ that resource p is available at time t based on two parameters, the average occupation time and the average vacancy time. Both parameters can be easily derived from empirical data.

Finally, the cost of each action (s, a) is considered as the travel cost $c_{(l, l')}$ for the edge (l, l') where l is the agent location in state s and l' is the node after taking action a . Obviously, the given model allows to compute proper policies as long as any resource does not converge to a state of being constantly occupied. In this case, returning infinitely often to the resource will result in finding the resource available at some point in time.

After defining all components to model resource search as an MDP, we now want to briefly describe the employed value iteration algorithm for computing the minimal expected cost $U^*(s)$ for each state $s \in S$. The idea of value iteration is to employ the Bellman equation to each state in each iteration until the utility for any state does not improve significantly anymore. The pseudocode of value iteration is presented in algorithm 1. After computing $U^*(s)$ processing the optimal policy is straight forward by applying the *argmin* function instead of the *min* function in the Bellman equation. Note that we employed value iteration due to its simplicity and for comparability with previous approaches. For a more detailed introduction to value iteration please refer to [7].

4 USING MDPS FOR RESOURCE SEARCH

A general problem of the method proposed above is the exponential increase of the state space with the considered resources P . Obviously, computing the basic value iteration method for large numbers on resources is not feasible. Thus, in order to make the approach applicable for real systems, we need to make sure that

Algorithm 1 Value iteration algorithm for computing the expected cost U of an optimal policy.

```

1: procedure VALUE ITERATION( $\text{mdp}, \epsilon$ )
2:   init  $U'$ 
3:   repeat
4:      $U \leftarrow U'$ 
5:      $\delta \leftarrow 0$ 
6:     for all  $s \in S$  do
7:        $U'[s] = \min_{a \in A(s)} c(s, a) + \gamma * \sum_{s'} P(s'|s, a) \cdot U[s']$ 
8:       if  $|U'[s] - U[s]| > \delta$  then
9:          $\delta \leftarrow |U'[s] - U[s]|$ 
10:    until  $\delta < \epsilon(1 - \gamma)/\gamma$ 
11:  return  $U$ 

```

the number of potential resources is as small as possible. Fortunately, in many cases the region where resources are acceptable is usually limited to a certain spatial region being specified by a user. For example, parking spots should be in walking distance to the actual target of the user. Similarly, charging stations should be close to planned route and in the region where the battery is already empty to make charging feasible. Similar statements hold for rental vehicles such as bikes and drop off stations. Thus, a first approach to limit the considered resources would be to compute isochrones around the actual target of the user and only consider resources within the tolerable walking distance. However, this approach would not work if the user would not specify an exact target location but only a rough area where he needs the resource. Additionally, in many cases the amount of acceptable resources would still push the computational effort beyond interactive query times. Thus, we propose another approach where MDPs are precomputed for typical target regions. For example, parking MDPs might be precomputed for typical shopping areas and MDPs for bike sharing make sense around universities. Let us note that precomputation can be done even for overlapping areas. Finally, the selection of the most suitable area can be easily performed by the user based on map interface displaying the available areas.

However, limiting the spatial area might not yield a successful reduction of available resources because many types of resources are spatially stacked on close-by locations. Examples are parking spots and rental bikes. For these type of resources, we propose to aggregate close-by resources and model them as a single aggregate resource in the MDP. For example, instead of modeling a road segment with ten parking spots as ten nodes having one resource, we aggregate all spots on the segment into a single aggregate resource. Of course, we have to adapt the probabilities for computing the likelihood $P(p(t) = 0)$ to mirror the fact that it is enough that any of the underlying resources is available. Thus, the probability $P(\hat{p}(t) = 0)$ of the aggregate resource \hat{p} for being available corresponds to the likelihood that not all of the underlying resources are occupied at time t . Formally, given a set of close-by resources $\hat{P} = \{p_1, \dots, p_k\}$ which are aggregated to the aggregate resource \hat{p} then $P(\hat{p}(t) = 0) = 1 - \prod_{p_i \in \hat{P}} P(p_i(t) = 1)$. However, $P(\hat{p}(t) = 1) = \prod_{p_i \in \hat{P}} P(p_i(t) = 1)$ because \hat{p} can only be considered as occupied if none of the underlying resources is available. A drawback of resource aggregation is that it removes the time for traveling between the aggregated resources. Thus, the learned policy might not be able to handle searches within the combined resources. However, we argue that with a proper

selection of resources, the user should be able to solve the problem by himself, e.g. finding a free resource among the combined resources. A heuristic, we employed to find sets of resources that can be aggregated is to make sure that all heuristic can be reached by making the same routing decision. In other words, all resources are placed on the same edge of the underlying road network.

A final aspect of our proposed method is the use of the learned policy in systems for guiding users to an available resource. As mentioned before, we precompute policies and let the user decide on the best suitable precomputed query region in order to find a suitable policy. Afterwards the system can combine the current user location and the online information about available resources to determine the current state and propose the direction aka action being stored in the policy. If the user wants to have an outlook on the following actions, the system could compute the most likely states for the current situation. However, in a volatile environment it is likely that these directions will change during travel to the next waypoint. A remaining task in this solution is to efficiently retrieve the optimal action for the current state for a given policy. Though we can identify the current state by combining location and resource information, we still have to find the state in the precomputed policy. Given the exponential size of the state space finding the entry for a particular state has to be done in an efficient way. Thus, we propose to store policies in a two dimensional array. We index the set of locations N by enumeration which indicates the first dimension. For the index of the second dimension, we encode the status information of the considered resources. This is done by mapping each resource to a particular bit in a bit vector of length $|P|$ and set the bit according to the availability of the resource. The resulting bit vector is then interpreted as an integer which indicates the second index in the array. Thus, the proposed action is retrieved in constant time.

5 EVALUATION

In order to evaluate our approach on a realistic scenario, we simulate the states of the parking spots for the city of Melbourne. The simulation samples for a given time-stamp and time interval the next states of any given parking spot. We have developed two different simulation models. The first model is based on the continuous-time Markov chain (similar to our MDP model) which enables us to do experiments based on the same conditions that our MDP assumes. However, continuous-time Markov chains might not provide a suitable model for real-world data. Thus, we implemented a simulator that is based on a real-world parking dataset, namely the freely available Melbourne ‘‘Parking bay arrivals and departures 2014’’ dataset¹. The dataset that contains arrival and departure times of most central business district parking bays. When we take the arrival and departure time into consideration, we can decide for every contained time-stamp, whether the parking bay is available or occupied. This simulation can be equated to a realistic real-time information system because it retrieves events that actually happened at the given time-stamp.

We compare our new approach denoted as D3RI to the UGCM algorithm that was proposed in [3] which also computes a policy but does not consider real-time information.

In figure 2, we can observe that the average time of finding a parking spot in the simulator is two to four times shorter

¹<https://data.melbourne.vic.gov.au/Transport-Movement/Parking-bay-arrivals-and-departures-2014/mq3i-cbxd>

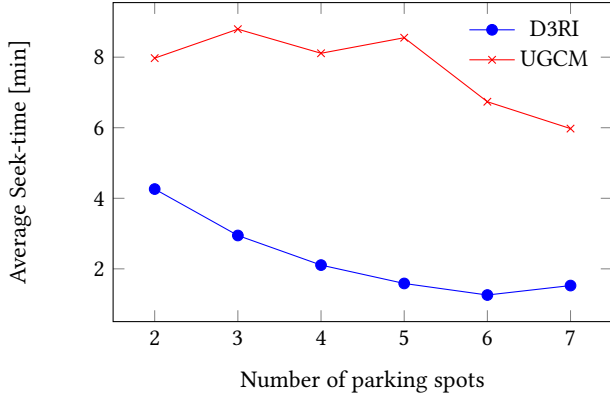


Figure 2: This figure shows the average search time of the agent in the time-continuous Markov chain simulator. Therefore we ran 1000 simulations and computed the average parking search time.

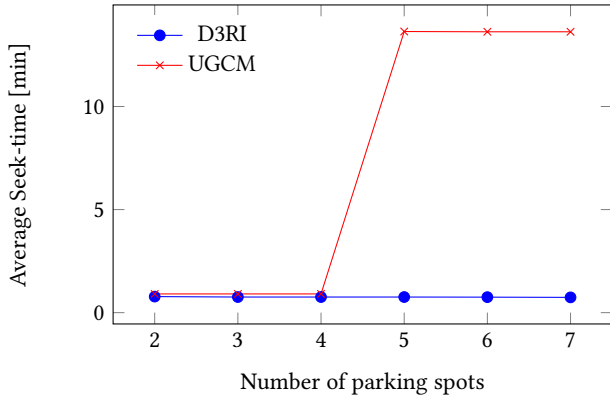


Figure 3: This figure shows the average search time of the agent in the Melbourne simulator. Therefore we ran 10 simulations on different days and computed the average parking search time.

with the approach proposed in this paper D3RI than the UGCM approach[3].

This is mainly because D3RI considers the real-time information of the parking spots while UGCM does not know if the parking spot, the agent is currently heading to, is available. This property can be observed in the real-world (Melbourne parking bays) simulation as well (c.f. figure 3). By knowing the exact states of the parking spots, D3RI can guide the driver directly to a freely available parking bay. However, if the information is not available, the driver may head to a nearby currently occupied parking spot. Because there are more possibilities for occupied parking spots, the approach of [3] becomes even worse when there are more spots available, since UGCM does not know which of the parking slots are free right now and usually with increasing amount of parking spots the amount of occupied spots increases as well. As expected, the use of real-time information yields a strong advantage which can be exploited by the D3RI model presented in this paper.

Figure 4 illustrates the runtime of our approach regarding the number of parking spots. In this experiment, we have used an error acceptance rate ϵ of 0.001 and a γ -value of 0.99. Obviously, the runtime increases exponentially with the amount of

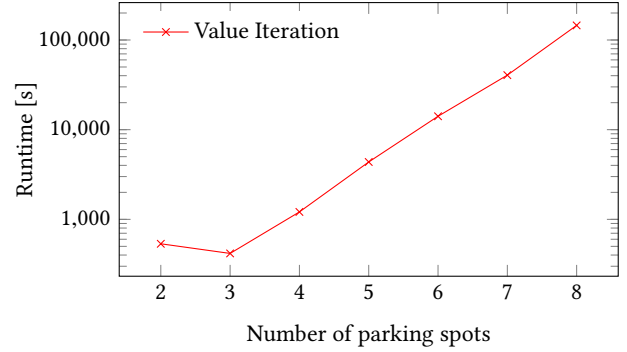


Figure 4: This figure shows the runtime of our approach depending on the number of spots. One may pay attention to the logarithmic y-axis.

considered parking spots. This suggests how essential resource aggregation is to reduce the computational overhead and the size of the policy. To conclude, without resource aggregation modeling realistic scenarios quickly get infeasible.

6 CONCLUSION

In this paper, we propose a novel approach for dynamic resource search in spatial networks considering real-time information on resource availability. To exploit valuable real-time information, solutions must react to recent developments in a flexible way. Therefore, we propose to learn a routing policy which provides the most promising action for any potentially occurring situation. To compute such a policy, we model resource search using real-time information as a Markov Decision Process (MDP). Furthermore, we discuss the design of a query system using pre-computed policies for guiding a user to an available resource. In our experiments, we compare our new approach using real-time information to a previous approach which computes policies based on historical data only. The results indicate that our new approach can exploit the real-time information to clearly outperform the comparison partner. For future work, we plan to examine techniques to mine areas for which building an MDP is most valuable. Furthermore, we will examine techniques to improve resource aggregation in order to compute MDPs for larger areas. Finally, we plan to examine database technology for storing and querying policies with very large state spaces.

REFERENCES

- [1] Andrew G Barto, Steven J Bradtke, and Satinder P Singh. 1995. Learning to act using real-time dynamic programming. *Artificial Intelligence* 72, 1 (1995), 81–138.
- [2] Blai Bonet and Hector Geffner. 2003. Labeled RTDP: Improving the Convergence of Real-Time Dynamic Programming. In *ICAPS*, Vol. 3. 12–21.
- [3] Qing Guo and Ouri Wolfson. 2016. Probabilistic spatio-temporal resource search. *GeoInformatica* (2016), 1–29.
- [4] Eric A Hansen and Shlomo Zilberstein. 2001. LAO*: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence* 129, 1-2 (2001), 35–62.
- [5] G. Jossé, K. A. Schmid, and M. Schubert. 2015. Probabilistic Resource Route Queries with Reappearance. In *Proceedings of the 18th International Conference on Extending Database Technology (EDBT15)*.
- [6] H Brendan McMahan, Maxim Likhachev, and Geoffrey J Gordon. 2005. Bounded real-time dynamic programming: RTDP with monotone upper bounds and performance guarantees. In *Proceedings of the 22nd international conference on Machine learning*. ACM, 569–576.
- [7] Stuart J. Russell and Peter Norvig. 2003. *Artificial Intelligence: A Modern Approach* (2 ed.). Pearson Education.
- [8] Scott Sanner, Robby Goetschalckx, Kurt Driessens, Guy Shani, et al. 2009. Bayesian Real-Time Dynamic Programming. In *IJCAI*. Citeseer, 1784–1789.
- [9] Trey Smith and Reid Simmons. 2006. Focused real-time dynamic programming for MDPs: Squeezing more out of a heuristic. In *AAAI*. 1227–1232.

Data Structures for Efficient Computation of Influence Maximization and Influence Estimation

Diana Popova
University of Victoria
British Columbia, Canada
dpopova@uvic.ca

Akshay Khot
University of Victoria
British Columbia, Canada
akshay03@uvic.ca

Alex Thomo
University of Victoria
British Columbia, Canada
thomo@uvic.ca

ABSTRACT

Algorithmic problems of computing *influence estimation* and *influence maximization* have been extensively studied for decades. We researched several data structures for implementing the Reverse Influence Sampling method proposed by Borgs, Brautbar, Chayes, and Lucier in 2014. Our implementations solve the problems of influence estimation and influence maximization in large graphs fast and using small memory footprint. For instance, we are able to produce results 3 times faster and scale 8 times more than a state-of-the-art algorithm, all this while preserving the theoretical guarantees of Borgs *et al.* for Reverse Influence Sampling.

1 INTRODUCTION AND DEFINITIONS

The most popular definition of *influence* relies on probabilistic reachability [4, 7]. The network is modeled as a directed graph and a node influence is calculated as the (expected) number of other nodes reachable from it. Given a set of *seeds* (initial nodes), *influence estimation* is calculated as the total number of nodes reachable from all the seeds in the set. To find a set of seeds that gives the maximum influence *spread* (the number of influenced nodes) is the *influence maximization* problem.

Running time and required space are the primary considerations for the algorithms solving influence estimation and influence maximization problems; the networks of interest are usually quite massive in size. Kempe *et al.* [7] showed that the influence maximization problem is NP-hard, and Chen *et al.* [3] showed for the influence estimation problem that computing the exact influence of a single seed is #P-hard. Moreover, as Feige [5] proved in 1998, the problem is hard to approximate to anything better than $1 - (1 - 1/s)^s$ of the optimum for a seed set of size s .

To model the influence spreading, Kempe *et al.* proposed the Independent Cascade (IC) model [7]: Starting from a seed, influence spreads in rounds/steps: each node after getting influenced has one possibility to influence its neighbors. IC selects edges from the seed neighborhood with *independent* probabilities. Influenced neighbors, in their turn, have one possibility to influence their neighbors forming a *cascade* of information propagation. Kempe *et al.* proved that influence maximization on the IC model is *monotone* and *submodular*, and therefore the approximate Greedy algorithm produces near-optimal solutions with a theoretical guarantee. Approximate Greedy starts with an empty seed set S . In each iteration, it adds to S a seed - the node with maximum *marginal* gain. IC became a standard model of information diffusion, and we are using it for our algorithms.

Building on the Kempe *et al.* results, several approximate algorithms with theoretical guarantees have been developed [3, 6, 10, 11, 13, 14]. However, the problem of scalability remains.

Recently, a new approach was proposed by Borgs *et al.* [2]: the Reverse Influence Sampling (RIS) method. RIS selects (uniformly at random, with replacement) a node and finds a set of nodes that *would have influenced* it. The set of found nodes is stored in a structure called *hypergraph*. This process is repeated many times. If a node appears often in sets of “influencers”, then this node is a good candidate for the most influential node in the graph. RIS is a faster algorithm for the influence maximization problem, obtaining the near-optimal approximation factor of $(1 - 1/e - \epsilon)$, for any $\epsilon > 0$, in time $O((m * k * \epsilon^{-2} * \log(n)))$, where k is the number of seeds. RIS can be modified to allow an early termination: if it is terminated after $O(\beta * m * k * \log(n))$ steps, then it returns a solution with an approximation factor that depends on β (the greater the β , the better the approximation is, and the guarantees are made precise in [2]). However, still RIS needs to sample nodes many times and consumes vast amounts of memory. The problem of scalability remains.

We note that there are several works that propose better bounds on the number of samples that need to be taken to achieve the same theoretical approximation (cf. [6, 9, 13]). Our research is orthogonal to these works. We aim at optimizing the computation and storage of sketches in the hypergraph; the aforementioned works aim at reducing the number of sketches needed.

Algorithms in this paper. Our main goal is to scale-up computing of influence maximization and influence estimation to large graphs with tens of millions of edges. We use several data structures all aiming at reducing the required memory and speeding up the computation.

- (1) We use Webgraph, a highly efficient, and actively maintained graph compression framework [1].
- (2) We design a new way of storing the hypergraph that significantly decreases the required space, without affecting the theoretical guarantee of the approximation.
- (3) We conduct experiments on large graphs on a consumer-grade laptop comparing the data structures, and provide a detailed analysis of the results.

2 PRELIMINARIES

Notations

Let $G = (V, E, p)$ be a directed graph, where V is the node set ($|V| = n$), E is the edge set ($|E| = m$), and $p : E \rightarrow [0, 1]$ is a probability function on the edges existence. Let S be a set of seeds. The influence spread of a seed set S under the Independent Cascade (IC) model, denoted by $\sigma(S)$, is defined as the expected total number of reachable nodes for S .

IM and IE Problems

Influence Estimation Problem (IE). Given a graph $G = (V, E, p)$ and a seed set $S \subseteq V$, compute the influence spread $\sigma(S)$ of S .

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

Influence Maximization Problem (IM). Given a graph $G = (V, E, p)$ and an integer k , find a seed set $S \subseteq V$ of size k that maximizes $\sigma(S)$.

3 PROPOSED DATA STRUCTURES

We developed three different algorithms implementing RIS. Each algorithm uses a distinct data structure for storing the hypergraph. We compared the performance of different data structures on a consumer-grade laptop.

3.1 RIS

The original RIS method (Algorithm 1 in [2]) selects nodes uniformly at random. Let v be such a node. Then RIS determines the set of nodes that *would* have influenced v by running a search in the *inverse* graph (the graph with the directions of edges reversed). RIS stores the found set of nodes, called a *sketch*, in a data structure, called a *hypergraph*. The process continues until the hypergraph reaches a pre-defined *weight*.

The weight of the hypergraph is defined as the number of graph edges "touched" by RIS. RIS selects an edge to follow at random, with a given edge probability p . During search, each edge incident to a visited node is counted as "touched" and contributes to the weight calculation. Note, that the edge is considered "touched" regardless of it being selected by the search or not. How large the weight should be in order to guarantee an approximation to the optimal solution is defined in [2].¹

The RIS hypergraph is a two-dimensional (2D) list that contains, for each node u in the graph, the IDs of the sketches where u appeared. Further, RIS runs an approximate Greedy algorithm on the hypergraph, which returns a set of k *seeds* (nodes with approximately maximal influence in the original graph).

3.2 Two-Dimensional List (2DL)

We started with a straightforward implementation of RIS described in subsection 3.1, where we use a two-dimensional list structure (list of lists) for storing the hypergraph. Algorithm 1 shows the pseudocode of this implementation.

For a better performance, we added the following improvements: 1. The Webgraph [1] format for the input inverse graph (saves space); 2. Java 8 parallel streams and lambda expressions (speeds up performance by executing several reachability procedures in parallel); 3. BitSet structure for marking deleted sketches (speeds up the marginal influence calculation); and 4. Leskovec *et al.* technique [8] (speeds up the seed calculation).

Influence estimation is part of seeds calculation. With minor changes, the code provides the solution for IE problem, when a seed set is inputted.

We compared the runtime and space of 2DL and DIM, a state-of-the-art implementation of RIS by Ohsaka *et al.* ([12]). For both algorithms, we used the same lower bound on the hypergraph weight from [2]. Our 2DL implementation significantly outperforms DIM. Testing results are discussed in detail in subsection 4.1.

3.3 Flat Arrays (FA)

FA implementation modifies the BuildHypergraph procedure of 2DL (subsection 3.2). The pseudocode is shown in Algorithm 2.

To store the hypergraph, FA creates two flat, one-dimensional, arrays of integers: *sketches* and *nodes*. *sketches* stores the sketch

¹Theorem 4.1 in [2], version 5, updated June 22, 2016.

Algorithm 1 2DL

Input: directed graph G with n nodes and m edges, coefficient β , number of seeds k

Output: seeds set $S \subseteq V$ of size k , spread $\sigma(S)$

```

1:  $R \leftarrow \beta * m * k * \log(n)$ 
2:  $H \leftarrow \text{BuildHypergraph}(R)$ 
3: return  $\text{GetSeeds}(H)$ 
4: procedure BUILDHYPERGRAPH( $R$ )
5:   while  $H\_weight < R$  do
6:      $v \leftarrow$  random vertex of  $G^T$ 
7:      $sketch \leftarrow$  reachable nodes in  $G^T$  starting from  $v$ 
8:     for each node  $u \in sketch$  do
9:       append  $sketchID$  to  $u$ 's list of sketches
10:       $count[u] \leftarrow count[u] + 1$ 
11:   return hypergraph  $H$ 
12: procedure GETSEEDS( $H$ )
13:    $S \leftarrow \emptyset$ ,  $\sigma(S) \leftarrow 0$ 
14:   for  $i = 1, \dots, k$  do
15:     seed  $v_i \leftarrow \text{argmax}_v \{count[v]\}$ 
16:      $S.insert(v_i)$ 
17:      $\sigma(S) \leftarrow \sigma(S) + count[v_i]$ 
18:     remove the sketches containing  $v_i$ 
19:   output  $S$ ,  $\sigma(S)$ 
```

ID, *nodes* stores the node IDs reached by this sketch. Arrays *sketches* and *nodes* are synchronized, so that knowing the index of a sketch, we can easily find the corresponding nodes, and *vice versa*. In the following figure we show an example of a *sketches* array (first row) and corresponding *nodes* array (second row):

0	0	0	0	1	1	2	2	3	4	4	5
0	1	2	3	2	3	1	3	2	2	3	0

In this example, sketch IDs and their corresponding node IDs are divided by double bars from other sketches: sketch 0 contains four nodes: 0, 1, 2, and 3; sketch 1 contains two nodes: 2 and 3; and so on. Note that sketches are listed in ascending order, and the corresponding nodes for each sketch are listed in ascending order as well. We use these features for speeding up the calculation of seeds. Testing FA on real-life graphs shows its better usage of space and faster performance than 2DL (subsection 4.1).

Algorithm 2 FA

```

1: procedure BUILDHYPERGRAPH( $R$ )
2:   initialize sketches and nodes arrays to -1
3:   while  $H\_weight < R$  do
4:      $v \leftarrow$  random vertex of  $G^T$ 
5:      $sketch \leftarrow$  reachable nodes in  $G^T$  starting from  $v$ 
6:     for each node  $u \in sketch$  do
7:        $sketches[i] \leftarrow sketchID$ 
8:        $nodes[i] \leftarrow u$ 
9:        $count[u] \leftarrow count[u] + 1$ 
10:   return hypergraph  $H = (sketches, nodes)$ 
```

3.4 Compressed Flat Arrays (CS-FA)

Here we present a more efficient implementation, the CS-FA algorithm (Algorithm 3). The main difference between CS-FA and FA is the design of the *sketches* array: CS-FA stores the accumulated

Algorithm 3 CS-FA

```
1: procedure BUILDHYPERGRAPH( $R$ )
2:   initialize  $nodes$  array to -1
3:   while  $H\_weight < R$  do
4:      $v \leftarrow$  random vertex of  $G^T$ 
5:      $sketch \leftarrow$  reachable nodes in  $G^T$  starting from  $v$ 
6:     for each node  $u \in sketch$  do
7:        $node\_count \leftarrow node\_count + 1$ 
8:       add  $nodeID$  to array  $nodes$ 
9:        $count[u] \leftarrow count[u] + 1$ 
10:    add  $node\_count$  to array  $sketches$ 
11:   return hypergraph  $H = (sketches, nodes)$ 
```

count of nodes included in sketches, thus making the *sketches* array compressed. Now we do not need to store sketch id's explicitly: sketch id's are the indexes in array *sketches*. The example below shows that sketch 0 includes three nodes, sketch 1 includes $(5 - 3) = 2$ nodes, and so on. The *nodes* array lists the corresponding nodes.

sketches:

3	5	6	8	10
---	---	---	---	----

nodes:

0	1	3	0	1	0	2	3	3	4
---	---	---	---	---	---	---	---	---	---

When we want to retrieve a sketch with id, say i , we need to find where its nodes start in the nodes array. This is given by the number stored in $sketches[i - 1]$ or 0 if $i = 0$. Testing shows CS-FA's smaller footprint and better run time than 2DL's or FA's (subsection 4.1).

4 EXPERIMENTAL RESULTS

We tested our IM and IE solutions by extensive experiments on several real-world graphs. For brevity, we included in this paper only the most interesting and telling results for IM and their analysis. All the presented results are achieved on a consumer-grade laptop with 16G of main memory.

We implemented the algorithms in Java 8 taking advantage of parallel streams and lambda expressions, and used Webgraph [1] as a graph compression framework. We compared our implementations with each other and with the DIM algorithm, implemented in C++ ([12]). We used the DIM code from <https://github.com/todo314/dynamic-influence-analysis>.

Datasets. Due to space constraints, we only present results for three real world graphs. Results for other datasets were similar. The datasets are available from the Laboratory for Web Algorithmics (<http://law.di.unimi.it/datasets.php>).

Dataset	n	m
UK100K	100,000	3,050,615
CNR-2000	325,557	3,216,152
EU-2005	862,664	19,235,140

Table 1: Datasets ordered by m .

While the size of the networks we considered is in the medium range, since each node can be sampled many times (we use sampling with replacement), the count of edges touched by the algorithms is in the billions. For example, the smallest of presented datasets, UK100K, requires a hypergraph with a weight of at least 5.6 billion, in order to produce the IM solution for $\beta = 16$ and $k = 100$.

Equipment. The experiments were conducted on a laptop with processor 2.2 GHz Intel Core i7 (4-core), RAM 16GB 1600 MHz DDR3, running OS X Yosemite.

Parameters. The parameters we use in our testing are as follows. k is the number of seeds in the seed set, β is a coefficient in Borgs *et al.* formula for the hypergraph weight, and p is the probability of edge existence. The tests are conducted varying k and β for $p = 0.1$.

4.1 Comparison of arrays, 2D list, and DIM performance

Fig. 1 shows the total running time and the time used for seeds calculation by DIM vs. our implementation of 2DL vs. FA vs. CS-FA. The test shown was conducted on CNR-2000, for $k = 10$, $p = 0.1$, and $\epsilon = 0.1$, varying β in powers of 2, from 2 to 128.

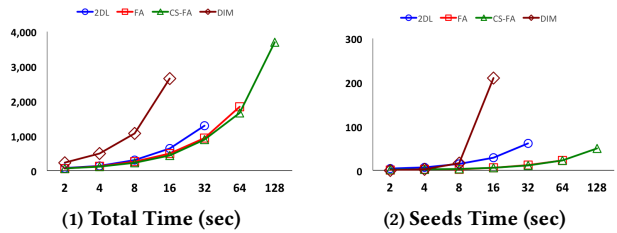


Figure 1: Processing time for cnr-2000; $k=10$, varying β .

The two-dimensional list implementations, DIM and 2DL, run slower and require more memory than array implementations. The reason for comparatively poor performance of 2D list implementations is the fragmentation of the main memory, when allocating space for each second-dimension list of sketch numbers for a node. This causes the memory manager to perform a lot of work trying to rearrange memory blocks. Improvements implemented in 2DL listed in subsection 3.2 allow for a better time performance on both the hypergraph computation and seed calculation, compared to DIM. For example, for $\beta = 16$, DIM took three times longer than 2DL to produce the result. The running times of FA and CS-FA are almost identical with each other. This is good for CS-FA; the compression we perform not only does not slow down CS-FA, but it makes CS-FA slightly faster due to better memory utilization. Both FA and CS-FA are faster than 2DL and DIM.

On both charts in Fig. 1, some data points are missing, because of the required memory being higher than what is available on the machine. 2DL and FA can handle runs with a β up to 32 and 64, respectively, while CS-FA can handle β equal to 128 due to its smaller memory footprint. That is, CS-FA scales the most, about 8 times more than DIM.

Fig. 2 shows the performance of 2DL, FA, and CS-FA when parameters k and β are growing, from the first chart, where all three implementations could run to completion, till the last one, where only the most efficient data structure (CS-FA) could produce one result, for the lowest β . The larger the β and k , the longer it takes for building the hypergraph and calculating the seeds, within one graph. This can be seen in the charts, while following a column from the top chart down. The larger the graph, the longer it takes for building the hypergraph and calculating the seeds. This can be seen in the charts, while following a row from the left chart to the right.

The largest hypergraph, successfully created and processed on the laptop, touched almost 5.6 billion edges. This hypergraph was

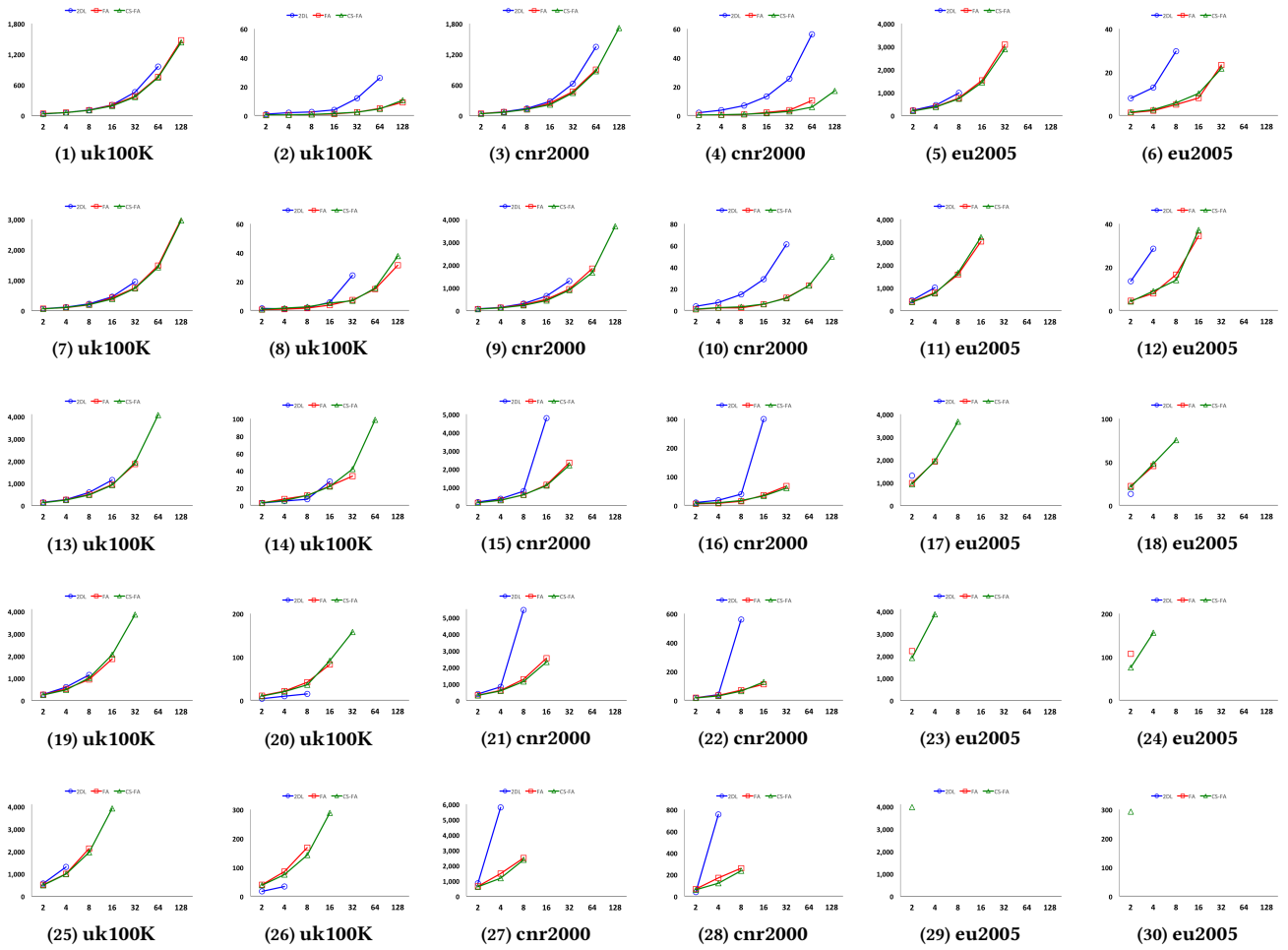


Figure 2: Total time (sec), and seeds time (sec). Per row, $k = 5, 10, 25, 50, 100$

processed by CS-FA (subsection 3.4). It took CS-FA one hour six minutes, including five minutes for calculating 100 seeds. We do not know another algorithm that can process such a hypergraph on a comparable machine.

Finally, in both Fig. 1 and 2, we observe that the time for calculating seeds is only a small part of the total time, which influenced our decision to not show separately experiments for IE (due to space constraints).

5 CONCLUSIONS AND FUTURE RESEARCH

We presented several implementations for computing influence estimation and influence maximization on graphs with multimillion edges. Our algorithms use different data structures. We tested the performance of these data structures on larger graphs, and provided a comparative analysis of test results. We substantially reduce the running time and required memory, without affecting the theoretical guarantees, to the point that multimillion-edge graphs could be processed on a consumer-grade laptop. Future research will involve further compression and parallelism aiming at scaling the computation of influence to bigger networks.

The source code for this paper can be found at: <https://github.com/dianapopova/InfluenceMax>

REFERENCES

- [1] P. Boldi and S. Vigna. The webgraph framework i: compression techniques. In *WWW*, 2004.
- [2] C. Borgs, M. Brautbar, J. Chayes, and B. Lucier. Maximizing social influence in nearly optimal time. In *SODA*, pages 946–957, 2014.
- [3] W. Chen, Y. Wang, and S. Yang. Efficient influence maximization in social networks. In *KDD*, pages 199–208, 2009.
- [4] E. Cohen, D. Delling, T. Pajor, and R. F. Werneck. Sketch-based influence maximization and computation: Scaling up with guarantees. In *CIKM*, pages 629–638, 2014.
- [5] U. Feige. A threshold of $\ln n$ for approximating set cover. *Journal of the ACM (JACM)*, 45(4):634–652, 1998.
- [6] K. Huang, S. Wang, G. S. Bevilacqua, X. Xiao, and L. V. S. Lakshmanan. Revisiting the stop-and-stare algorithms for influence maximization. *PVLDB*, 10:913–924, 2017.
- [7] D. Kempe, J. Kleinberg, and É. Tardos. Maximizing the spread of influence through a social network. In *KDD*, pages 137–146, 2003.
- [8] J. Leskovec, A. Krause, C. Guestrin, C. Faloutsos, J. VanBriesen, and N. Glance. Cost-effective outbreak detection in networks. In *KDD*, pages 420–429, 2007.
- [9] H. T. Nguyen, M. T. Thai, and T. N. Dinh. Stop-and-stare: Optimal sampling algorithms for viral marketing in billion-scale networks. In *SIGMOD*, pages 695–710, 2016.
- [10] H. T. Nguyen, M. T. Thai, and T. N. Dinh. A billion-scale approximation algorithm for maximizing benefit in viral marketing. *IEEE/ACM Trans. Netw.*, 25(4):2419–2429, Aug. 2017.
- [11] N. Ohsaka, T. Akiba, Y. Yoshida, and K.-i. Kawarabayashi. Fast and accurate influence maximization on large networks with pruned monte-carlo simulations. In *AAAI*, pages 138–144, 2014.
- [12] N. Ohsaka, T. Akiba, Y. Yoshida, and K.-i. Kawarabayashi. Dynamic influence analysis in evolving networks. *PVLDB*, 9(12):1077–1088, 2016.
- [13] Y. Tang, Y. Shi, and X. Xiao. Influence maximization in near-linear time: A martingale approach. *SIGMOD '15*, New York, NY, USA.
- [14] Y. Tang, X. Xiao, and Y. Shi. Influence maximization: Near-optimal time complexity meets practical efficiency. *SIGMOD '14*, New York, NY, USA.

A Roadmap towards Declarative Similarity Queries

Nikolaus Augsten
 University of Salzburg
 Salzburg, Austria
 nikolaus.augsten@sbg.ac.at

ABSTRACT

Despite many research efforts, similarity queries are still poorly supported by current systems. We analyze the main stream research in processing similarity queries and argue that a general-purpose query processor for similarity queries is required. We identify three goals for the evaluation of similarity queries (declarative, efficient, combinable) and identify the main research challenges that must be solved to achieve these goals.

1 INTRODUCTION

In a similarity query, two data objects “match” if they are similar. Similarity queries are required in scenarios where equality and exact matches are not effective, for example, when dealing with noisy data (e.g., in data analytics, ETL processes, data cleaning, and entity resolution), for detecting small differences between data objects (e.g., finding similar molecular structures in computational biology), for comparing complex objects (e.g., trees, graphs, or multimedia content), and for querying physical measurement data (e.g., time series, spatial objects, sensor data).

Similarity queries involve a *similarity function* defined on a specific data type and an *operator*. The similarity function assesses the similarity (or dissimilarity) between pairs of objects, e.g., the edit distance between strings or trees, Cosine similarity between sets, or Euclidean distance between vectors. The operator defines the input signature (e.g., lookup vs. join) and the pairs that qualify for the result set (e.g., top-*k*, range, or skyline query).

Database management systems (DBMS) for *exact queries* (i.e., predicates based on equality, less-than, or greater-than) have evolved into powerful and mature systems, which transparently and efficiently deal with data storage and querying. Unfortunately, this development has not happened for similarity queries. Applications that require advanced similarity features cannot rely on general-purpose systems that transparently handle data storage and querying. Instead, similarity queries must be dealt with in custom, ad hoc code. Writing custom code for similarity queries is expensive, requires advanced query processing skills, and often results in inflexible, hard-coded query plans. Changing the query beyond simple parameter settings requires additional programming efforts.

In this paper we argue that the integration of similarity queries into declarative DBMS (relational or non-relational) and the efficient processing in a systems context are the next challenges to be solved for similarity queries. In the past, the main research focus was on physical operators and access methods: new evaluation algorithms and index structures for specific combinations of operators, data types, and similarity functions were proposed. The evaluation of similarity queries in a larger systems context, however, has received little attention.

Developing a general-purpose query processor for similarity queries is challenging, both from a conceptual and a technical point of view. From a conceptual point of view, similarity queries pose a particular challenge. While the meaning of similarity is highly application dependent, the query interface should be general and serve a wide range of application needs. Note that application dependency is much less pronounced in equality queries: for most data types, the notion of equality is well defined and application independent.

From a technical point of view, many techniques that are effective for equality queries are not applicable to similarity queries. Techniques for equality queries often rely on exact matches or some ordering, which cannot be assumed between similar objects. An example is hashing, which leverages the fact that identical data values are hashed to the same bucket. This does not hold for similar values: they will typically be hashed to different buckets. Another example is sorting, which is not reliable for similar data items since even a small change may have a large impact on the position of the data item in the sort order. Further, little is known about the interaction of physical similarity operators with other, equality-based operators in the query context. This involves all aspects of query processing, including modeling similarity operators at the logical level (e.g., extending relational algebra), rewriting query plans, estimating the cost of similarity operators, and gracefully adapting to limited memory resources.

This paper suggests to depart from the main stream in similarity research with its narrow focus on individual physical operators and studies similarity queries from a broader systems perspective. The overall goal is to develop a deep understanding of all aspects of similarity queries that are required to build a general-purpose query processor for these queries.

2 DEC – DESIDERATA FOR SIMILARITY QUERIES

We identify three core requirements for a similarity query processing system: *declarative, efficient, combinable (DEC)*, i.e., declarative queries that combine equality and similarity predicates should be processed efficiently. We believe that all DEC requirements must be satisfied in a useful end-to-end system for similarity queries. We next discuss each of the three DEC requirements, which are orthogonal aspects of a query answering system.

Declarative. The queries should be declarative, i.e., the query describes *what* the answer to the query should look like rather than *how* the answer should be computed. A declarative approach allows flexible queries and a clearer separation between logical and physical layer. While the users express their queries at the logical level, the system must translate the query into a physical execution plan. Declarative data query languages are the predominant approach in the traditional relational model [9] with SQL (Structured Query Language) as a practical query language, but have also been applied to non-relational data models (e.g., XPath¹ for semi-structured data) and to cluster computing

¹<http://www.w3.org/TR/xpath-30/>

systems with flexible data types (e.g., HiveQL [25] for analytic queries on large data clusters). Effective techniques for translating declarative queries to physical operators are required, and the resulting query plans must be optimized.

Efficient. The queries should be executed efficiently, i.e., efficient query plans that consider appropriate techniques for similarity operators should be constructed. Similarity predicates [13, 27] often involve complex and expensive functions, and a straightforward evaluation of the similarity predicate on each tuple is not feasible. A rich set of algorithms and access methods have been proposed to allow similarity queries to be processed efficiently. Such techniques are often based on an index or a filter/verify approach to reduce the number of expensive predicate evaluations [6, 11, 23]. Filters produce a set of candidates which contains false positives; in the verification step the false positives are removed. A widespread approach avoids the evaluation of the cross product in similarity joins by rewriting the join into an equality join on tokens, e.g., q -grams in a string similarity join [13]. When the similarity function is a metric, the triangle inequality and metric index structures can be leveraged [26]. A system for similarity queries should be able to effectively apply the efficient evaluation strategies that have been developed.

Combinable. Similarity and equality predicates should be arbitrarily combinable into complex queries. In useful queries, similarity predicates are embedded into a larger query context which includes a mix of equality and similarity predicates. In order to evaluate such a complex query efficiently, the query must be considered as a whole. It is not enough to process the similarity part independently from the equality part and then intersect (or union) the results. This may lead to very poor evaluation strategies since large intermediate results are produced. An example are conjunctive queries in which each individual predicate has weak selectivity (i.e., produces a large intermediate result set), but the conjunction of all predicates is strongly selective (i.e., the final result is small). A query processor for similarity queries should not be limited to the evaluation of the similarity predicate of the query, but should be able to evaluate both similarity and exact predicates alike.

3 SIMILARITY SUPPORT IN CURRENT SYSTEMS

Database Management Systems. DBMS typically offer basic support for similarity queries on strings, e.g., Soundex, a phonetic transcription of English surnames. More advanced similarity predicates are supported in the form of user defined functions (UDF). The UDF is used to evaluate the similarity predicate on a pair of attribute values. Unfortunately, the DBMS cannot produce efficient evaluation plans for queries with UDFs since they are a black box for the optimizer. UDFs are typically applied in a naive way (e.g., on each pair of tuples in a join [13]). Since the DBMS does not understand the properties of the similarity join, efficient filter/verify techniques (which have been proposed for similarity joins) or other optimizations cannot be leveraged. Overall, DBMS offer declarative, combinable similarity queries, but fail to process them efficiently.

Custom Software. Applications that require more sophisticated algorithms rely on custom software, e.g., as part of an entity resolution tool [10], at the back-end of a search form [18], or in some scientific application [7]. Due to the narrow focus and the predictable nature of the queries, the query plan is generated by

hand and hard-coded, and appropriate algorithms and indexes are implemented. Hard-coded query plans are problematic since the quality of a query plan depends on the query parameters and the data distribution. Good query plans are particularly important for similarity queries, which often involve expensive predicates [14, 16, 27]. Extending custom software with new queries requires substantial programming efforts.

Integrating Similarity into DBMS. There have been several attempts to extend DBMS with similarity features. Barioni et al. [2] propose the SIREN system and an SQL extension to deal with similarity queries over multimedia and relational data. SIREN processes the similarity part outside the DBMS in a separate system and integrates the results in a second step. This separation substantially limits the options for efficient query plans since the similarity predicate cannot be freely moved in the query plan. Guliato et al. [15] propose an extension for PostgreSQL (an open source DBMS) for image retrieval. Similarly, the `pg_similarity` extension of PostgreSQL defines a set of similarity functions (e.g., edit and q -gram distance for strings). These approaches do not change the query processor, but are UDF-based and cannot leverage advanced algorithms, indexes, and optimization techniques. Silva et al. [21] integrate physical operators for similarity join and group-by into the core of PostgreSQL; the similarity operators are limited to numeric values. The Metric Similarity Search Implementation Framework (MESSIF) [3] is a library for object retrieval in metric space; it supports metric indexes and algorithms for range queries and top- k selection, but no declarative query interface or a query optimizer. An attempt to define an SQL-based query language for MESSIF has been reported, but query processing is not discussed.

Silva et al. [22] study the conceptual evaluation and query transformation rules for various types of similarity queries based on metric distances. In addition to the well-known ϵ -join (range distance join), the k NN-join (k nearest neighbor join), and k D-join (k -distance join) they also discuss join-around, a combination of range and nearest neighbor join. In terms of select queries, ϵ - and k NN-selection are discussed. Carey and Kossmann [5] and Bruno et al. [4] discuss the optimization of top- k queries.

The system closest to our vision is DIMA [24], which extends SQL with range queries over strings and sets. DIMA builds on Spark, supports distributed query evaluation, and uses a signature-based approach to distribute the query load and filter candidate matches. Compared to our vision discussed in the next section, the high-level similarity operators are not split into algebraic primitives, there is no metadata to select filters and transform the queries accordingly, and a high-level similarity operator is mapped one-to-one to the respective physical operator.

Other Systems. Entity resolution systems like NADEEF [10, 12] use similarity functions between individual attribute values to deal with noise in the data. Digital libraries deal with mixed objects (multimedia, text, 3D structures) with the goal of preserving digital objects, allowing users to enter new items, and accessing content. In both cases, the query patterns are hard-coded in the application, i.e., declarative queries are not supported.

Information retrieval systems like INDRI² or Lucene³ store collections of documents (e.g., plain text, HTML, PDF) in files, build indexes over these files, and deal with stemming and stop word removal. Queries are phrases, possibly with wildcards, that

²<http://www.lemurproject.org/indri>

³<http://lucene.apache.org>

can be combined with boolean operators; the search can be limited to individual fields of a document (e.g., the title field). The query result is a list of documents, which is ranked by relevance. Similarity queries are supported at a very basic level. Lucene, for instance, supports the phonetic encodings Soundex and Metaphone, and edit distance selection. The queries in these systems are limited to selection and ranking; more complex query patterns (e.g., joins) are not supported.

4 ROADMAP

Challenges. We identify three challenges that must be addressed to build similarity queries into declarative database management systems: (1) A new, *minimal set of algebraic operators* for similarity queries must be defined. (2) *Dynamic rewriting*: metadata about eligible filters, indexes, and data transformations must be made available to the query optimizer. (3) A *uniform cost model* for physical similarity operators must be developed.

Minimal algebra. The goal is to develop a new algebra for similarity queries which extends relational algebra with a *minimal set of operator primitives* that is able to express a wide range of similarity operators.

Similarity (unlike equality) is not a binary predicate, and data items can be ranked by their “degree” of similarity w.r.t. some query. This gives rise to a large variety of new matching principles, for example, k -closest neighbor selection, similarity group-by, or join-around (k closest neighbors within a maximum distance range). Previous work, e.g., Silva et al. [22], defines an algebraic operator for each of these matching principles. We believe that this approach will not scale: (a) Each new operator requires deep changes in the system. (b) Query rewriting rules for each pair of operators must be defined, leading to a quadratic number of such rules. The key to success will be to establish a minimal, non-redundant set of primitives that are composed to express high-level operators. This will provide great flexibility in reordering queries, and introducing new high-level operators will not require changes in the algebra.

The new algebra should satisfy the following requirements. (a) *Minimal*: the new operators should be small, non-redundant primitives; the new algebra should cleanly separate two orthogonal concepts, which have often been mixed in previous work: the similarity function between two objects (e.g., edit distance) and the matching principle (e.g., top- k join). (b) *Expressive*: a wide range of similarity queries should be expressible in the new algebra; complex similarity operators (e.g., top- k join with edit distance), for which efficient implementations exist at the physical level, may be expressed by composing several of the new primitives at the logical level. (c) *Extendable*: Introducing new physical operators or query flavors at the user level should not require changes in the algebra. (d) *Transformable*: The new algebra should provide equivalence rules which allow the optimizer to reorder the logical operators in a flexible way. This is particularly important for small primitives since multiple primitives may compose a single physical operator.

Dynamic Rewriting. In order to expand queries with filters, metadata about similarity functions (e.g., edit distance), their relationship and properties (e.g., upper and lower bounds, metric properties), and applicable filters (e.g., q -grams for range joins) must be available. This metadata will be leveraged to dynamically produce query transformation rules. New filters can easily be introduced by updating the filter ontology: no changes in the optimizer are required.

The metadata stores properties of (a) similarity functions, (b) matching principles, and (c) filters, and their relationships. Similarity functions may satisfy metric properties (e.g., string edit distance) or even some L_i -norm (e.g., cardinality of set intersection, geographic distances). The relationships between similarity functions are guarantees like lower and upper bounds (e.g., the q -gram distance provides a lower bound for the more expensive edit distance). The filter ontology must further relate similarity function / matching principle pairs to eligible filter techniques. Some filters require data transformations, e.g., the computation of tokens [1], which should be specified in the metadata. The filter ontology must also provide information about the selectivity of filters, which will be used to model the query cost. Given a similarity predicate, the ontology should be able to derive all eligible filters (including necessary transformations and selectivity).

Expanding queries with filters leads to very different physical plans for a single logical query. The physical plans may involve techniques that have been developed by different communities, e.g., metric and token-based techniques. A uniform cost model is required to evaluate the plans. The cost model must consider the cost of similarity functions, which may be very expensive, filter selectivity, and the effect of filters on the data distribution.

Uniform cost model. A new cost model for physical similarity operators must be developed. The cost model should quantify the cost of different physical query plans, which result among others from introducing filters into the query plan or transforming data to the appropriate representation (e.g., string data may be transformed into tokens or signatures for filtering purposes). In the past, cost models for some individual operations (e.g., M-tree lookups [8]) have been developed. For other operations (e.g., set similarity joins [19]) experimental studies provide qualitative insights, but lack a model to predict the cost. Selectivity estimates [17, 20] are an important input for cost models, but selectivities are independent of physical operators. A quantitative assessment of the costs of all physical operators in the query is required. Computing comparable cost estimations is particularly challenging for approaches that have evolved in different communities, for example, edit similarity, token-based approaches, and metric techniques. The cost estimation will need to take into account the data distribution, any query parameters, the available resources, and the filter selectivities. The cost model should further integrate well with existing cost models for non-similarity operators since the overall query cost must be assessed.

Query processing. We envision the evaluation of a query that includes similarity predicates as illustrated in Figure 1: The parser generates a query tree that involves both standard relational operators and the new algebraic similarity primitives. Thereby, a high-level similarity operator like a top- k join or a skyline query will be represented by a number of low-level algebra operators. The query planner consults the similarity metadata to learn about eligible filter techniques like lower and upper bounds for the given similarity function and operator. Thereby, the query planner is not limited to the high-level operators in the original query. For example, there may only be filter information for nearest neighbor queries in the metadata, but the query is join-around [22] (which combines nearest neighbor and range join). The planner decomposes join around into algebraic primitives and tries to rearrange and match the primitives to known combinations in the metadata.

The query plans with filters will typically include additional algebra operators (representing the filter). Some filters will require

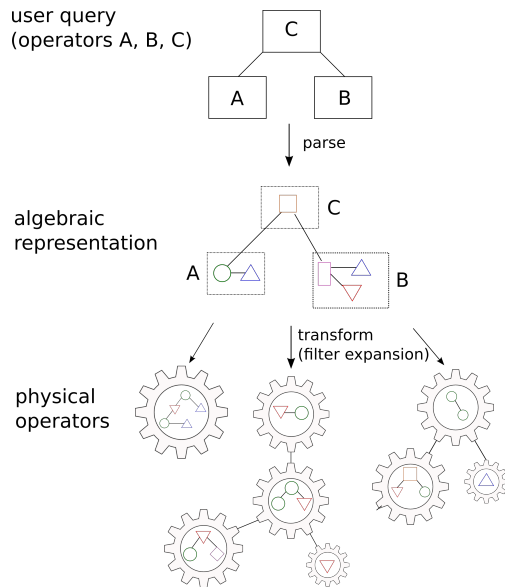


Figure 1: Interaction of system components.

on-the-fly index construction (e.g., prefix index for set similarity joins) or data transformations (e.g., tokens or signatures). The transformation rules for expanding the query plan with filters will be dynamically derived from the metadata. When new filters are registered in the metadata, they will trigger new query plans that apply these filters.

Finally, the cost of the query plans must be assessed. To this end, the algebraic operators must be arranged such that they match existing physical operators. Note that the logical query level (e.g., a query expressed in an SQL-like language) and the physical level are independent. There is no one-to-one match between the operators visible to the users and the physical operators actually implemented in the system. Rather, the operators in the user query are disassembled into algebra, and subtrees of the query plans are matched against existing physical operators.

5 CONCLUSION

This paper suggests to depart from the main stream in similarity research with its narrow focus on individual physical operators and studies similarity queries from a broader systems perspective. The overall goal is to develop a deep understanding of all aspects of similarity queries that are required to build a general-purpose query processor for these queries. The systems aspects discussed in this paper must be solved to enable wide applicability and impact of similarity queries in real systems. The main research challenges are the development of a minimal algebra for similarity queries, the design and querying of metadata regarding filter techniques for similarity queries, and the cost estimation for physical similarity operators.

A declarative interface will have a fundamental impact on the user interaction with similarity queries. Database users will no longer need to write ad-hoc code for evaluating similarity predicates. Instead, similarity predicates are expressed in a declarative way and are processed efficiently. Application developers will not need to bother about the details of similarity query processing. We expect a general-purpose query processor to trigger a wide adoption of similarity queries also in applications that so far could not afford the overhead of writing custom code.

Finally, a declarative similarity query processor will set new standards in the research community. New algorithms for physical operators must be evaluated against the query plans produced by the similarity-enabled optimizer, which can leverage a wide range of techniques, and dynamically adapts to query parameters and data distribution. Further, new algorithm proposals will not only be measured by their performance in an isolated setting, but also by their usefulness in a systems context.

Acknowledgements. This work was partially supported by the Austrian Science Fund (FWF): P 29859-N31.

REFERENCES

- [1] Nikolaus Augsten, Armando Miraglia, Thomas Neumann, and Alfons Kemper. 2014. On-the-Fly Token Similarity Joins in Relational Databases. In *ACM SIGMOD*.
- [2] Maria C. N. Barioni, Humberto Razente, Agma Traina, and Caetano Traina Jr. 2006. SIREN: A Similarity Retrieval Engine for Complex Data. In *Proc. Int. Conf. VLDB*.
- [3] Michal Batko, David Novak, and Pavel Zezula. 2007. MESSIF: Metric Similarity Search Implementation Framework. In *Digital Libraries: Research and Development*. Vol. 4877. Springer Berlin Heidelberg.
- [4] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. 2002. Top-k Selection Queries over Relational Databases: Mapping Strategies and Performance Evaluation. *ACM Trans. Database Syst.* 27, 2 (2002).
- [5] Michael J. Carey and Donald Kossmann. 1998. Reducing the Braking Distance of an SQL Query Engine. In *Proc. Int. Conf. VLDB*.
- [6] Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. 2006. A Primitive Operator for Similarity Joins in Data Cleaning. In *Proc. Int. Conf. IEEE ICDE*.
- [7] Davide Chicco and Marco Masseroli. 2015. Software Suite for Gene and Protein Annotation Prediction and Similarity Search. *IEEE/ACM Trans. on Computational Biology and Bioinformatics* 12, 4 (2015).
- [8] Paolo Ciaccia, Marco Patella, and Pavel Zezula. 1998. A Cost Model for Similarity Queries in Metric Spaces. In *Int. Proc. ACM PODS*.
- [9] Edgar F. Codd. 1970. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM* 13, 6 (1970).
- [10] Michele Dallachiesa, Amr Ebaid, Ahmed Eldawy, Ahmed Elmagarmid, Ihab F. Ilyas, Mourad Ouzzani, and Nan Tang. 2013. NADEEF: A Commodity Data Cleaning System. In *ACM SIGMOD*.
- [11] Dong Deng, Guoliang Li, Jianhua Feng, and Wen-Syan Li. 2013. Top-k String Similarity Search with Edit-Distance Constraints. In *Proc. Int. Conf. IEEE ICDE*.
- [12] Ahmed Elmagarmid, Ihab F. Ilyas, Mourad Ouzzani, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Si Yin. 2014. NADEEF/ER: Generic and Interactive Entity Resolution. In *ACM SIGMOD*.
- [13] Luis Gravano, Panagiotis G. Ipeirotis, H. V. Jagadish, Nick Koudas, S. Muthukrishnan, and Divesh Srivastava. 2001. Approximate String Joins in a Database (Almost) for Free. In *Proc. Int. Conf. VLDB*.
- [14] Sudipto Guha, H. V. Jagadish, Nick Koudas, Divesh Srivastava, and Ting Yu. 2002. Approximate XML Joins. In *ACM SIGMOD*.
- [15] Denise Gulati, Ernani V. de Melo, Rangaraj M. Rangayyan, and Robson C. Soares. 2009. POSTGRESQL-IE: An Image-Handling Extension for PostgreSQL. *J. Digital Imaging* 22, 2 (2009).
- [16] Yu Jiang, Guoliang Li, Jianhua Feng, and Wen-Syan Li. 2014. String Similarity Joins: An Experimental Evaluation. *PVLDB* 7, 8 (2014).
- [17] Hongrae Lee, Raymond T. Ng, and Kyuseok Shim. 2011. Similarity Join Size Estimation Using Locality Sensitive Hashing. *PVLDB* 4, 6 (2011).
- [18] Guoliang Li, Shengyue Ji, Chen Li, and Jianhua Feng. 2009. Efficient Type-Ahead Search on Relational Data: A TASTIER Approach. In *ACM SIGMOD*.
- [19] Willi Mann, Nikolaus Augsten, and Panagiotis Boursos. 2016. An Empirical Evaluation of Set Similarity Join Techniques. *PVLDB* 9, 9 (2016).
- [20] Arturas Mazeika, Michael H. Böhlen, Nick Koudas, and Divesh Srivastava. 2007. Estimating the Selectivity of Approximate String Queries. *ACM Trans. Database Syst.* 32 (2007).
- [21] Yasin N. Silva, Ahmed M. Aly, Walid G. Aref, and Per-Ake Larson. 2010. SimDB: A Similarity-Aware Database System. In *ACM SIGMOD*.
- [22] Yasin N. Silva, Walid G. Aref, Per-Ake Larson, Spencer Pearson, and Mohamed H. Ali. 2013. Similarity Queries: Their Conceptual Evaluation, Transformations, and Processing. *PVLDB* 22, 3 (2013).
- [23] Yasin N. Silva, Spencer Pearson, and Jason A. Cheney. 2013. Database Similarity Join for Metric Spaces. In *Proc. Int. Conf. Similarity Search and Applications*.
- [24] Ji Sun, Zeyuan Shang, Guoliang Li, Dong Deng, and Zhifeng Bao. 2017. Dima: A Distributed In-Memory Similarity-Based Query Processing System. *PVLDB* 10, 12 (2017).
- [25] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive: A Warehousing Solution over a Map-Reduce Framework. *PVLDB* 2, 2 (2009).
- [26] Pavel Zezula, Giuseppe Amato, Vlastislav Dohnal, and Michal Batko. 2006. *Similarity Search—The Metric Space Approach*. Advances in Database Systems, Vol. 32. Springer.
- [27] Xiang Zhao, Chuan Xiao, Xuemin Lin, and Wei Wang. 2012. Efficient Graph Similarity Joins with Edit Distance Constraints. In *Proc. Int. Conf. IEEE ICDE*.

Interactive Exploration of Composite Items

Sihem Amer-Yahia
 CNRS, Univ. Grenoble Alpes
 Saint Martin D'Hères, France
 sihem.amer-yahia@cnrs.fr

Senjuti Basu Roy
 New Jersey Institute of Technology
 Newark, NJ, USA
 senjuti.basuroy@njit.edu

ABSTRACT

Data exploration is seeing a renewed interest in our community. With the rise of big data analytics, this area is growing to encompass not only approaches and algorithms to find the next best data items to explore but also interactivity, i.e. accounting for feedback from the data scientist during the exploration. Interactivity is essential to account for evolving needs during the exploration and also customize the discovery process. In this tutorial, we focus on the interactive exploration of Composite Items (CIs).

CIs address complex information needs and are prevalent in online shopping where products are bundled together to provide discounts, in travel itinerary recommendation where points of interest in a city are combined into a single travel package, and task assignment in crowdsourcing where personalized micro-tasks are composed and recommended to workers. CI formation is usually expressed as a constrained optimization problem. For instance, in online shopping, package retrieval can retrieve the cheapest smartphones (optimization objective) with compatible accessories (constraints). Similarly, a city tour must be the most popular and conform to a total time and cost budget. A data scientist interested in exploring a variety of CIs has to repeatedly reformulate optimization problems with new constraints and objectives. In this tutorial, we investigate the applicability of interactive data exploration approaches to CI formation.

We will first review CI applications and shapes (15mn). We then discuss three big research questions (60mn): (i) algorithms for CI formation, (ii) modes of exploration for CIs, and (iii) human-in-the-loop CIs. We will conclude with research directions (15mn).

The proposed tutorial is timely. It brings together several related efforts and addresses unsolved questions in the emerging area of human-in-the-loop exploration of complex information needs. The tutorial is relevant to the general area of data science and more specifically to Scalable Analytics, Data Mining, Clustering and Knowledge Discovery, Indexing, Query Processing and Optimization, and Crowdsourcing. The technical topics covered are constrained optimization, ranking semantics, clustering, algorithms, and empirical evaluations.

1 OUTLINE AND SCOPE

1.1 Scope

The tutorial targets theoreticians and practitioners interested in the development of data science applications. It should be of particular interest to an audience who wants to learn about how different domains, such as product recommendation, scientific simulation, or team formation in the social sciences, have been developing their “siloed” definitions of CIs. Tutorial attendees are expected to have basic knowledge in algorithms and data management. Knowledge in constrained optimization is not necessary.



Figure 1: A Star CI

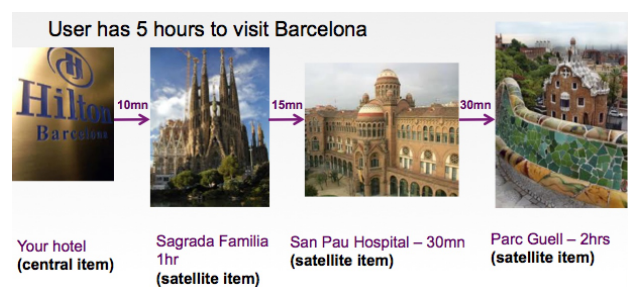


Figure 2: A Chain CI

1.2 Outline

1.2.1 CI Applications and Shapes (15mn). We will begin the tutorial by providing example applications that justify the need for Composite Items (CIs). This part will gather those examples and attempt to unify them. Figures 1, 2, 3 show the many shapes CIs can take in shopping (star shape, where satellite items must be compatible with a central item), traveling (chain shape, where an item must be geographically close to its preceding item), and dining (where items are jointly co-reviewed by the same people), respectively.

1.2.2 Research Questions: (60mn). As our tutorial topic is *building and exploring CIs interactively*, the second part of the tutorial is organized into three big research questions: (i) algorithms for CI retrieval, (ii) how data exploration is typically done for large-scale datasets and its applicability to exploring CIs, and (iii) what types of user interactivity are common and their applicability to building and exploring CIs interactively. For each question, we will review the state of the art and discuss new research challenges.

Research Question 1: CI Retrieval (20mn). Different CI shapes require the specification of different constraints and optimizations, thereby leading to a no “one-size-fits-all” CI definition. We will discuss why and how the nuances of data, such as type heterogeneity, dimensionality, distribution, or even storage, impact

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

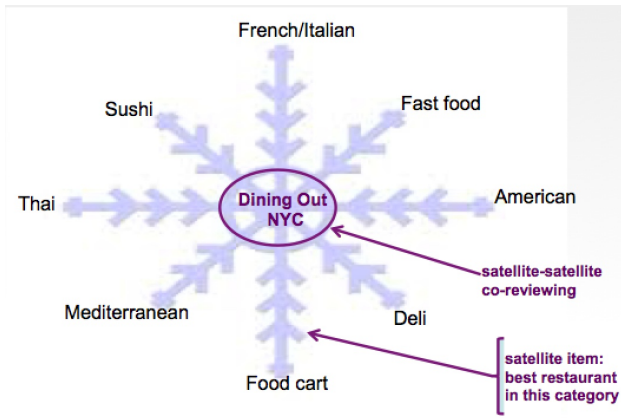


Figure 3: A Snowflake CI

the semantics and performance of building different shapes of CIs. For example, the type of aggregation that a climatologist seeks to build a CI that reflects a meaningful “climate model” is significantly different from that of a retail business manager who packages compatible items for product recommendation, or from a crowdsourcing platform that bundles diverse micro-tasks to address workers’ boredom. Using these applications, we will present scenarios where exploration of CIs and interactivity are essential. For instance, in the domain of experiment design, we will show the necessity of leveraging feedback from domain scientists to select a different set of parameters that appropriately capture a scientific simulation process, and represent it as a CI. In the travel world, we will argue that interactivity helps refine one’s partial needs and build personalized packages [20, 24]. Similarly, in crowdsourcing, a CI may represent a bundle of diverse tasks that are more exciting to workers than a ranked list of tasks [4].

Research Question 2: CI Exploration (20mn). The aim of this part is to bridge the gap between exploration in emerging data science applications and exploring structured data, and discuss how that can serve CI exploration.

We will first describe approaches that are popularly used among data scientists to explore large-scale datasets. Such data exploration techniques lack well-defined objectives and are mostly done following a trial-and-error approach. Consequently, most of the visualization-based data exploration techniques that data scientists popularly use are ad-hoc and unprincipled.

After that we will discuss data exploration techniques that are more principled and investigated in conjunction with structured databases, especially considering a user input, in the form of queries, example data, or data distributions. Some notable examples in that space relate to faceted search and query expansion techniques [8, 9, 11, 12, 14, 16, 21, 29, 30]. We will also discuss some recent work that investigates exploration and visualization techniques intended to assist users by looking for similar data distributions [5, 22, 23, 26] or in an example-driven exploration [15, 19]. We will argue why these techniques are not directly applicable to CI exploration.

Finally, we will discuss why CI exploration needs to go beyond existing techniques and rely on optimization-guided data exploration as in [18]. We will pose open problems and computational challenges in designing appropriate solutions for exploring CIs.

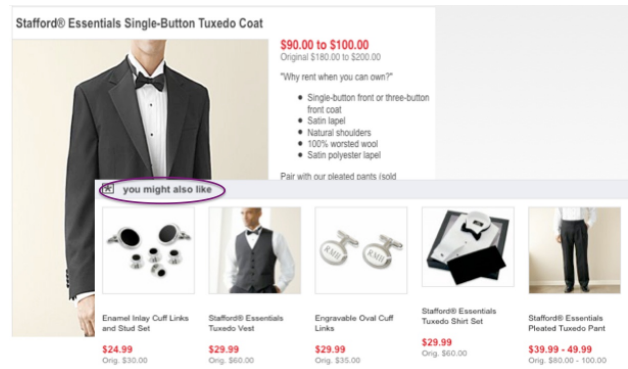


Figure 4: Recommending Satellite Items

Research Question 3: Human-in-the-Loop CIs (20mn). The third challenge is how to incorporate human-in-the-loop effectively to enable interactive data exploration.

We will first discuss different types of interactivity that a user is allowed to provide, ranging from binary responses to capturing implicit actions. We will do that in the context of different contexts such as crowdsourcing, recommender systems, experiment design, or machine learning applications that require supervised samples for training models. In conjunction, we will discuss different types of users, such as naive users or domain experts, and investigate what types of challenges, such as expressivity of interaction, bias, and real-time interactions, they incur. We will examine how these questions can be revisited in the context of building CIs and review preliminary work on satellite item recommendation [7, 20] (also see Figure 4), and on adding or deleting specific items in CIs [24] (also see Figure 5). An additional challenge when interacting with CIs is the visual layout. Unlike “flat” items, aiding users in their interactions with CIs, via recommendations or maps for instance, is necessary for a full-fledged exploration.

In a second part, we will describe other types of feedback that are rather non-traditional and only discussed in recent work, such as feedback on metadata rather than on the entire object. We will discuss how such feedback is processed for answering queries, feature selection, or feature engineering in the data science pipeline [10, 17, 32]. We will describe our vision on hybrid approaches that discuss how to leverage sophisticated yet limited human feedback in the computational loop and show their utility for CIs.

1.3 Research Challenges: (15mn)

In this last part of the tutorial, we will summarize and brainstorm our overarching framework to enable optimization-guided data exploration techniques that enable a human-in-the-loop approach. We will discuss what types of applications it will support and conclude by outlining some major challenges in combining CI exploration and interactive CIs.

1.4 Overlap with Previously Presented Tutorials

In WWW 2015, the authors presented a tutorial on composite items in the context of complex crowdsourcing [1]. This tutorial will have a very small overlap with that tutorial in the part that reviews various CI definitions. Other than that, the content of this tutorial was not presented before.

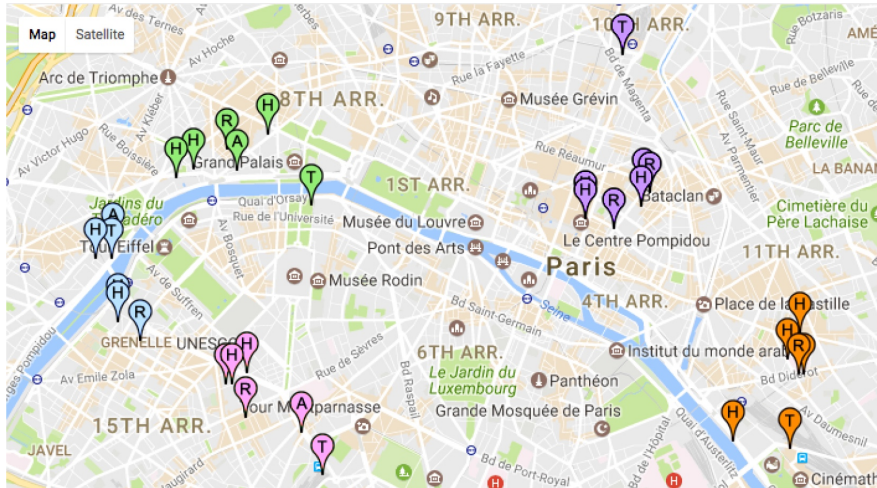


Figure 5: CIs on a Map

2 BIOGRAPHY

The authors published seminal papers on composite item retrieval and data exploration.

Sihem Amer-Yahia, *sihem.amer-yahia@cnsr.fr*, is a Research Director in Grenoble. Her interests are at the intersection of data management and social data exploration. She is the Editor-in-Chief of the VLDB Journal for Europe and Africa and PC co-chair of PVLDB 2018 and WWW Tutorials 2018.

Senjuti Basu Roy, *senjuti.basuroy@njit.edu*, is an Assistant Professor at NJIT. Her research interests lie in the area of data and content management of web and structured data with a focus on exploration, analytics, and algorithms. She is the PC Co-chair of SIGMOD 2018 mentorship track, PC co-chair of VLDB 2018 PhD workshop track.

REFERENCES

- [1] Sihem Amer-Yahia and Senjuti Basu Roy. 2015. From Complex Object Exploration to Complex Crowdsourcing.. In *Proceedings of the 24th International Conference on World Wide Web*. ACM, 1531–1532.
- [2] Sihem Amer-Yahia, Francesco Bonchi, Carlos Castillo, Esteban Feuerstein, Isabel Mendez-Díaz, and Paula Zabala. 2013. Complexity and algorithms for composite retrieval. In *Proceedings of the 22nd International Conference on World Wide Web*. ACM, 79–80.
- [3] Sihem Amer-Yahia, Francesco Bonchi, Carlos Castillo, Esteban Feuerstein, Isabel Mendez-Díaz, and Paula Zabala. 2014. Composite retrieval of diverse and complementary bundles. *IEEE Transactions on Knowledge and Data Engineering* 26, 11 (2014), 2662–2675.
- [4] Sihem Amer-Yahia, Éric Gaussier, Vincent Leroy, Julien Pilourdault, Ria Mae Borromeo, and Motomichi Toyama. 2016. Task Composition in Crowdsourcing. In *IEEE International Conference on Data Science and Advanced Analytics, DSAA, Montreal, QC, Canada, October 17-19, 2016*. 194–203.
- [5] Sihem Amer-Yahia, Sofia Kleisarchaki, Naresh Kumar Kolloju, Laks V. S. Lakshmanan, and Ruben H. Zamar. 2017. Exploring Rated Datasets with Rating Maps. In *Proceedings of the 26th International Conference on World Wide Web, WWW 2017, Perth, Australia, April 3-7, 2017*. 1411–1419.
- [6] Sihem Amer-Yahia, Laks Lakshmanan, and Cong Yu. 2009. Socialscope: Enabling information discovery on social content sites. *arXiv preprint arXiv:0909.2058* (2009).
- [7] Senjuti Basu Roy, Sihem Amer-Yahia, Ashish Chawla, Gautam Das, and Cong Yu. 2010. Constructing and exploring composite items. In *ACM SIGMOD*. ACM, 843–854.
- [8] Senjuti Basu Roy, Haidong Wang, Gautam Das, Ullas Nambiar, and Mukesh Mohania. 2008. Minimum-effort driven dynamic faceted search in structured databases. In *Proceedings of the 17th ACM conference on Information and knowledge management*. ACM, 13–22.
- [9] Adriane Chapman and HV Jagadish. 2009. Why not?. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. ACM, 523–534.
- [10] Justin Cheng and Michael S Bernstein. 2015. Flock: Hybrid crowd-machine learning classifiers. In *Proceedings of the 18th ACM Conference on Computer*

Supported Cooperative Work & Social Computing. ACM, 600–611.

- [11] Kyriaki Dimitriadou, Olga Papaemmanouil, and Yanlei Diao. 2014. Explore-by-example: An automatic query steering framework for interactive data exploration. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 517–528.
- [12] Zhian He and Eric Lo. 2014. Answering why-not questions on top-k queries. *IEEE Transactions on Knowledge and Data Engineering* 26, 6 (2014), 1300–1315.
- [13] Arlind Kopliku, Karen Pinel-Sauvagnat, and Mohand Boughanem. 2014. Aggregated search: A new information retrieval paradigm. *ACM Computing Surveys (CSUR)* 46, 3 (2014), 41.
- [14] Chengkai Li, Ning Yan, Senjuti B Roy, Lekhendro Lisham, and Gautam Das. 2010. Facetedpedia: dynamic generation of query-dependent faceted interfaces for wikipedia. In *Proceedings of the 19th international conference on World wide web*. ACM, 651–660.
- [15] Davide Mottin, Matteo Lissandrini, Yannis Velegarakis, and Themis Palpanas. 2017. New Trends on Exploratory Methods for Data Analytics. *PVLDB* 10, 12 (2017), 1977–1980.
- [16] Davide Mottin, Alice Marascu, Senjuti Basu Roy, Gautam Das, Themis Palpanas, and Yannis Velegarakis. 2013. A probabilistic optimization framework for the empty-answer problem. *Proceedings of the VLDB Endowment* 6, 14 (2013), 1762–1773.
- [17] Besmira Nushi, Adish Singla, Andreas Krause, and Donald Kossmann. 2016. Learning and Feature Selection under Budget Constraints in Crowdsourcing. In *AAAI Conference on Human Computation and Crowdsourcing (HCOMP)*.
- [18] Behrooz Omidvar-Tehrani, Sihem Amer-Yahia, and Alexandre Termier. 2015. Interactive User Group Analysis. In *Proceedings of the 24th ACM International Conference on Information and Knowledge Management, CIKM 2015, Melbourne, VIC, Australia, October 19 - 23, 2015*. 403–412.
- [19] Olga Papaemmanouil, Yanlei Diao, Kyriaki Dimitriadou, and Liping Peng. 2016. Interactive Data Exploration via Machine Learning Models. *IEEE Data Eng. Bull.* 39, 4 (2016), 38–49. <http://sites.computer.org/debull/A16dec/p38.pdf>
- [20] Senjuti Basu Roy, Gautam Das, Sihem Amer-Yahia, and Cong Yu. 2011. Interactive itinerary planning. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*. IEEE, 15–26.
- [21] Senjuti Basu Roy, Haidong Wang, Ullas Nambiar, Gautam Das, and Mukesh Mohania. 2009. Dynacet: Building dynamic faceted search systems over databases. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*. IEEE, 1463–1466.
- [22] Tarique Siddiqui, John Lee, Albert Kim, Edward Xue, Xiaofu Yu, Sean Zou, Lijin Guo, Changfeng Liu, Chaoran Wang, Karrie Karahalios, et al. 2017. Fast-Forwarding to Desired Visualizations with Zenvisage.. In *CIDR*.
- [23] Tarique Siddiqui, John Lee, Albert Kim, Edward Xue, Xiaofu Yu, Sean Zou, Lijin Guo, Changfeng Liu, Chaoran Wang, Karrie Karahalios, and Aditya G. Parameswaran. 2017. Fast-Forwarding to Desired Visualizations with Zenvisage. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminate, CA, USA, January 8-11, 2017, Online Proceedings*.
- [24] Manish Singh, Ria Mae Borromeo, Anas Hosami, Sihem Amer-Yahia, and Shady Elbassuoni. 2017. Customizing Travel Packages with Interactive Composite Items. In *2017 IEEE International Conference on Data Science and Advanced Analytics, DSAA 2017, Tokyo, Japan, October 16-18, 2017*.
- [25] Quoc Trung Tran and Chee-Yong Chan. 2010. How to conquer why-not questions. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 15–26.
- [26] Manasi Vartak, Samuel Madden, Aditya Parameswaran, and Neoklis Polyzotis. 2014. SEEDB: automatically generating query visualizations. *Proceedings of the VLDB Endowment* 7, 13 (2014), 1581–1584.

- [27] Min Xie, Laks VS Lakshmanan, and Peter T Wood. 2010. Breaking out of the box of recommendations: from items to packages. In *Proceedings of the fourth ACM conference on Recommender systems*. ACM, 151–158.
- [28] Min Xie, Laks VS Lakshmanan, and Peter T Wood. 2011. Comprec-trip: A composite recommendation system for travel planning. In *IEEE International Conference on Data Engineering (ICDE)*. 1352–1355.
- [29] Ning Yan, Chengkai Li, Senjuti B Roy, Rakesh Ramegowda, and Gautam Das. 2010. Facetedpedia: enabling query-dependent faceted search for wikipedia. In *Proceedings of the 19th ACM international conference on Information and knowledge management*. ACM, 1927–1928.
- [30] Sivan Yogevev, Haggai Roitman, David Carmel, and Naama Zwerdling. 2012. Towards expressive exploratory search over entity-relationship data. In *Proceedings of the 21st International Conference on World Wide Web*. ACM, 83–92.
- [31] Nan Zhang, Chengkai Li, Naemul Hassan, Sundaresan Rajasekaran, and Gautam Das. 2014. On skyline groups. *IEEE Transactions on Knowledge and Data Engineering* 26, 4 (2014), 942–956.
- [32] James Y Zou, Kamalika Chaudhuri, and Adam Tauman Kalai. 2015. Crowd-sourcing feature discovery via adaptively chosen comparisons. *arXiv preprint arXiv:1504.00064* (2015).

Recent Advances in Recommender Systems: Matrices, Bandits, and Blenders

Georgia Koutrika
Athena Research Center
Athens, Greece
georgia@imis.athena-innovation.gr

ABSTRACT

Recent years have witnessed an explosion in methods applied to solve the recommendation problem. Modern recommender systems have become increasingly more complex compared to their early content-based and collaborative filtering versions. In this tutorial, we will cover recent advances in recommendation methods, focusing on matrix factorization, multi-armed bandits, and methods for blending recommendations. We will also describe evaluation techniques, and outline open issues and challenges. The ultimate goal of this tutorial is to present a toolkit of new recommendation methods in perspective to data-related problems, and highlight opportunities and new research paths for researchers and practitioners that work on problems in the intersection of recommender systems and databases.

1 INTRODUCTION

The proliferation of digital content in a plurality of forms (including e-news, movies, and online courses), along with the popularity of portable devices has created immense opportunities as well as challenges for systems to provide users with information and services that best serve the users' needs. Matching consumers with the most appropriate items is key to enhancing user satisfaction and loyalty. Recommender systems come to the rescue providing advice on movies, products, travel, leisure activities, and many other topics. Personalized recommendations can elevate the user experience. That is why e-commerce leaders like Amazon and Netflix have made recommender technology a salient part of their systems [15].

Broadly speaking, recommender systems are based on one of two strategies. *Content-based filtering* creates a profile for each user or item to characterize their features. The profiles allow the recommender system to associate users with matching items. An alternative to content-based filtering relies only on past user behavior (e.g., previous purchases or user ratings). This approach is known as *collaborative filtering*, a term coined by the developers of Tapestry, an early recommender system [10]. Collaborative filtering analyzes relationships between users and interdependencies among items to identify new user-item associations based on which to make recommendations.

Recent years have witnessed an explosion in methods applied to solve the recommendation problem and modern recommender systems have become increasingly more complex. Matrix factorization methods, popularized with the Netflix prize [15], have become a dominant methodology within collaborative filtering recommenders due to their superior performance both in terms of recommendation quality and scalability. On the other hand,

multi-armed bandits are becoming popular in interactive recommendation settings, for example for recommending songs in Pandora [29]. Ranked lists of items generated by different recommender systems are blended together into the final list of recommendations shown to the user. The blending problem is essentially a multi-objective optimization problem, with objectives such as relevancy, coverage and diversity competing with each other [6, 25]. Overall, the landscape of recommender systems has changed immensely since the first content-based and collaborative filtering systems emerged, and the state-of-the-art approaches show outstanding results and open up new opportunities and research paths.

Interestingly, we are used to thinking of recommendations in the context of systems that serve items, such as movies, products, friend connections, etc, to users. The reality is that the recommendation problem arises in many different scenarios beyond those targeting user consumption. For example, in the context of databases, such scenarios include but are not limited to data exploration, query optimization, visualization, data integration, and workflow design, where the purpose is to select tuples [7], queries [9], views [8], exploration actions [19], query plans [30], visualization graphs [26–28], work flows [12], and so forth. While there is work in these areas, it pales compared to the amount and diversity of recommendation methods developed for items such as movies and products.

Therefore, the purpose of this tutorial is two-fold. First, it aims at providing a comprehensive overview of recent advances in recommendation methods, highlighting their capabilities and their impact. The focus of the tutorial is on matrix factorization methods, multi-armed bandits, and blending methods. It will discuss major techniques, evaluation methodologies, and open issues. Since the recommendation problem appears in many different settings, it is the purpose of this tutorial to provide a solid framework for placing novel recommendation work into perspective for data-related problems, provide a toolkit of new methods, and highlight research opportunities for researchers and practitioners in database systems, data-intensive applications, and the intersection of recommender systems and databases.

The following sections describe the structure and contents of the tutorial. The tutorial does not require any prior knowledge in recommender systems since there will be detailed introductions to the relevant techniques.

2 OUTLINE

The tutorial is structured in the following parts.

2.1 Recommendation Framework

The objective of this section is to introduce the audience to the recommendation problem, define the basic concepts as well as the different instances of the problem (e.g., rating prediction and whole-page optimization), and provide an overview of the classical approaches for generating recommendations.

Recommender systems appeared back in the nineties, and two broad categories of recommendation approaches emerged: content-based and collaborative filtering. Content-based approaches analyze user past selections (e.g., web pages they visited, movies they watched) to learn user preferences and recommend items with similar content to the user’s past selections and likes.

Collaborative filtering analyzes usage data (e.g., user ratings and purchases) and recommends to the user either items with similar usage characteristics as the items selected by this user or items from users with similar usage characteristics to this user.

We will explain the basic characteristics and operations behind each family of methods as well as their advantages and their shortcomings. The objective of this section is to lay the necessary foundations for the rest of the tutorial. It also aims at preparing the ground for understanding the methods to be presented subsequently and their impact.

2.2 Matrix Factorization

Matrix Factorization has gained popularity in recommender systems in recent years due to its superior performance both in terms of recommendation quality and scalability. Part of its success is due to the Netflix Prize contest for movie recommendations, which popularized a Singular Value Decomposition (SVD) based matrix factorization algorithm [13]. The Netflix Prize competition began in October 2006 and has fueled much recent progress in the field of collaborative filtering. For the first time, the research community gained access to a large-scale, industrial strength data set of 100 million movie ratings while the nature of the competition encouraged rapid development, where innovators built on each generation of techniques to improve prediction accuracy.

Experience with datasets such as the Netflix Prize data has shown that matrix factorization methods deliver accuracy superior to classical nearest-neighbor techniques [14]. At the same time, they offer a compact memory-efficient model that systems can learn relatively easily. What makes these techniques even more convenient is that models can integrate naturally many crucial aspects of the data, such as multiple forms of user feedback, temporal dynamics, and confidence levels.

In its basic form, matrix factorization characterizes both items and users by vectors of latent factors inferred from the ratings users gave to the items. Early systems use Singular Value Decomposition (SVD) – a well-established technique for identifying latent semantic factors in information retrieval – as a matrix factorization method for collaborative filtering. In the following years, several extensions to matrix factorization have been proposed and matrix factorization becomes the foundation in most recent recommender systems.

We will start with some background on Singular Value Decomposition and describe how early works (e.g., [23]) use SVD to capture latent relationships between customers and products and to produce a low-dimensional representation of the original customer-product space in order to compute the predicted likeliness of a certain product by a customer. We will introduce Low-rank Matrix Factorization [13, 21] followed by most recent important extensions to Matrix Factorization for recommendations, such as SLIM [20].

2.3 Multi-armed Bandits

Traditional recommender systems, including collaborative filtering, content-based filtering and hybrid approaches, can provide meaningful recommendations at an individual level by leveraging

users’ interests as demonstrated by their past activity. However, in many web-based scenarios (e.g., filtering news articles or display of advertisements), the content universe undergoes frequent changes, with content popularity changing over time as well. Furthermore, a significant number of visitors are likely to be entirely new with no historical consumption record whatsoever; this is known as a cold-start situation. These issues make traditional recommender approaches difficult to apply. In such highly dynamic recommendation domains, it is essential for the recommendation method to adapt to the shifting preference patterns of the users and the evolving space of items. Exploration-exploitation methods, a.k.a. multi-armed bandits, have been shown to be an excellent solution.

In probability theory, the multi-armed bandit problem is a problem in which a gambler at a row of slot machines (sometimes known as “one-armed bandits”) has to decide which machines to play, how many times to play each machine and in which order to play them. When played, each machine provides a random reward from a probability distribution specific to that machine. The objective of the gambler is to maximize the sum of rewards earned through a sequence of lever pulls.

For example [17], in the context of article recommendation, we may view articles in the pool as arms. When a presented article is clicked, a reward of 1 is incurred; otherwise, the reward is 0. With this definition of reward, the expected reward of an article is precisely its clickthrough rate (CTR), and choosing an article with maximum CTR is equivalent to maximizing the expected number of clicks from users, which in turn is the same as maximizing the total expected reward in the bandit formulation. Furthermore, we may “summarize” users and articles by a set of informative features that describe them compactly. By doing so, a bandit algorithm can generalize CTR information from one article/user to another, and learn to choose good articles more quickly, especially for new users and articles.

We will first present context-free K-armed bandit algorithms [4, 22], such as ϵ -greedy and upper confidence-bound (UCB) algorithms [1, 3, 16], and then move to contextual bandit algorithms [17, 29]. We will focus on multi-armed bandits used in the context of recommender systems and in particular in three problems: (a) Popularity ranking, to balance exposure of new items (exploration) with old winners (exploitation), (b) Model-based collaborative filtering [11, 18], and (c) Dueling bandits, to efficiently compare multiple recommendation methods [5, 24].

2.4 Blending Models

Several domains require “blending” of recommendations from different sources. Blending allows different recommendation strategies to develop independently, and combine their outputs post-hoc into a meta-recommender. The result aims at providing recommendations of higher quality and diversity. For instance, the Pinterest Homefeed is a personalized feed of content (i.e., pins) drawn from many sources, including followed users, followed topics, and recommendations, among other sources. Each type of content is ranked by its own specialized machine learning model, and then blended with a ratio-based round-robin method to create the final Homefeed [6].

We will examine different methods to blend recommendations, including fixed ratio, greedy, calibrated ranker and multi-armed bandit-approaches [2, 6, 25]. As we examine these blending systems, new questions arise as to how to measure success. Unlike traditional search ranking problems, recommender systems face

both short- and long-term optimization challenges as there is a need to balance immediate user-engagement metrics and long term ecosystem health. We will examine new such metrics and approaches to this end.

2.5 Lessons Learnt and Open Issues

In this section, we will discuss lessons learnt, open issues and new research directions created by these novel recommendation methods. We will also discuss about recommendation problems that do not target user consumption. In particular, we will describe such scenarios including data exploration, query optimization, visualization, data integration, workflow design, and so forth, where the purpose is to sort out not movies or products but queries, views, exploration actions, query plans, visualization graphs, and so forth. We will examine these problems in the light of the recent developments in the recommendation arena and discuss new research directions and opportunities that arise.

3 PRESENTER BIO

Georgia Koutrika is Director of Research at Athena Research Center in Greece. She has worked at HP Labs in Palo Alto, at IBM Research-Almaden in San Jose, and as a postdoctoral researcher at the Computer Science Dept., Stanford University. She has worked extensively on personalization and recommender systems, large-scale information extraction, entity resolution and information integration, and querying and data exploration interfaces. Her work has been incorporated in commercial products, has been described in 7 granted patents and 19 patent applications in the US and worldwide, and has been published in more than 80 research papers in top-tier conferences and journals. An IEEE Senior member, ACM member, and ACM SIGMOD Associate Information Director, Georgia has also served as a General Co-Chair for ACM SIGMOD 2016, Industrial Track PC Chair for EDBT 2016, and Workshop and Tutorial Co-Chair for IEEE ICDE 2016. She is currently Demo PC co-chair for ACM SIGMOD 2018.

REFERENCES

- [1] Rajeev Agrawal. Sample mean based index policies with $o(\log n)$ regret for the multi-armed bandit problem. *Advances in Applied Probability*, 27(4):1054–1078, 1995.
- [2] Amr Ahmed, Choon Hui Teo, S.V.N. Vishwanathan, and Alex Smola. Fair and balanced: Learning to present news stories. In *the Fifth ACM International Conference on Web Search and Data Mining*, WSDM '12, pages 333–342, New York, NY, USA, 2012. ACM.
- [3] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2-3):235–256, 2002.
- [4] Don Berry and Bert Fristedt. *Bandit Problems: Sequential Allocation of Experiments*, volume Monographs on Statistics and Applied Probability. Chapman and Hall, 1985.
- [5] Bangrui Chen and Peter I. Frazier. Dueling bandits with weak regret. *CoRR*, abs/1706.04304, 2017.
- [6] Stephanie deWet. Personalized content blending in the pinterest homefeed. In *2017 Netflix Workshop on Personalization, Recommendation and Search*, 2017.
- [7] Marina Drosou and Evaggelia Pitoura. Ymaldb: exploring relational databases via result-driven recommendations. *The VLDB Journal*, 22(6):849–874, Dec 2013.
- [8] Humaira Ehsan, Mohamed A. Sharaf, and Panos K. Chrysanthis. Muve: Efficient multi-objective view recommendation for visual data exploration. In *the 32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*, pages 731–742, 2016.
- [9] Magdalini Eirinaki, Suju Abraham, Neoklis Polyzotis, and Naushin Shaikh. Querye: Collaborative database exploration. *IEEE Trans. Knowl. Data Eng.*, 26(7):1778–1790, 2014.
- [10] David Goldberg, David A. Nichols, Brian M. Oki, and Douglas B. Terry. Using collaborative filtering to weave an information tapestry. *Commun. ACM*, 35(12):61–70, 1992.
- [11] Frédéric Guillou, Romaric Gaudel, and Philippe Preux. Collaborative filtering as a multi-armed bandit. In *the Workshop: Machine Learning for eCommerce*, NIPS'15, 2015.
- [12] Dietmar Jannach, Michael Jugovac, and Lukas Lerche. Supporting the design of machine learning workflows with a recommendation system. *Tiis*, 6(1):8:1–8:35, 2016.
- [13] Yehuda Koren. Factorization meets the neighborhood: A multifaceted collaborative filtering model. In *the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '08, pages 426–434, New York, NY, USA, 2008. ACM.
- [14] Yehuda Koren and Robert Bell. *Advances in Collaborative Filtering*, pages 145–186. Springer US, Boston, MA, 2011.
- [15] Yehuda Koren, Robert M. Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *IEEE Computer*, 42(8):30–37, 2009.
- [16] T. L. Lai and H. Robbins. Asymptotically efficient adaptive allocation rules. *Advances in Applied Mathematics*, 6(1):4–22, 1985.
- [17] Lihong Li, Wei Chu, John Langford, and Robert E. Schapire. A contextual-bandit approach to personalized news article recommendation. In *the 19th International Conference on World Wide Web*, WWW '10, pages 661–670, New York, NY, USA, 2010. ACM.
- [18] Shuai Li, Claudio Gentile, Alexandros Karatzoglou, and Giovanni Zappella. Data-dependent clustering in exploration-exploitation algorithms. *CoRR*, abs/1502.03473, 2015.
- [19] Tova Milo and Amit Somech. React: Context-sensitive recommendations for data analysis. In *the 2016 International Conference on Management of Data*, SIGMOD '16, pages 2137–2140, New York, NY, USA, 2016. ACM.
- [20] Xia Ning and George Karypis. Slim: Sparse linear methods for top-n recommender systems. In *ICDM*, 2011.
- [21] Arkadiusz Paterek. Improving regularized singular value decomposition for collaborative filtering. *KDD Cup and Workshop*, 2007.
- [22] Herbert Robbins. Some aspects of the sequential design of experiments. *Bulletin of the American Mathematical Society*, 58(5):527–535, 1952.
- [23] Badrul M. Sarwar, George Karypis, Joseph A. Konstan, and John T. Riedl. Application of dimensionality reduction in recommender system - a case study. In *the KDD Workshop on Web Mining for e-Commerce: Challenges and Opportunities (WebKDD)*, 2000.
- [24] Yanan Sui, Vincent Zhuang, Joel W. Burdick, and Yisong Yue. Multi-dueling bandits with dependent arms. *CoRR*, abs/1705.00253, 2017.
- [25] Choon Hui Teo, Houssam Nassif, Daniel Hill, Sriram Srinivasan, Mitchell Goodman, Vijai Mohan, and S.V.N. Vishwanathan. Adaptive, personalized diversity for visual discovery. In *the 10th ACM Conference on Recommender Systems*, RecSys '16, pages 35–38, New York, NY, USA, 2016. ACM.
- [26] Manasi Vartak, Silu Huang, Tarique Siddiqui, Samuel Madden, and Aditya G. Parameswaran. Towards visualization recommendation systems. *SIGMOD Record*, 45(4):34–39, 2016.
- [27] Manasi Vartak, Sajjadur Rahman, Samuel Madden, Aditya G. Parameswaran, and Neoklis Polyzotis. SEEDB: efficient data-driven visualization recommendation to support visual analytics. *PVLDB*, 8(13):2182–2193, 2015.
- [28] Kanit Wongsuphasawat, Dominik Moritz, Anushka Anand, Jock Mackinlay, Bill Howe, and Jeffrey Heer. Towards a general-purpose query language for visualization recommendation. In *Workshop on Human-In-the-Loop Data Analytics*, HILDA '16, pages 4:1–4:6, New York, NY, USA, 2016. ACM.
- [29] Tao Ye and Mohit Singh. Combining relevancy scoring and contextual bandits for personalized ranking in music discovery. In *the 2017 Netflix Workshop on Personalization, Recommendation and Search*, 2017.
- [30] Jihad Zahir and Abderrahim El Qadi. A recommendation system for execution plans using machine learning. *Mathematical and Computational Applications*, 21(2), 2016.

openCypher: New Directions in Property Graph Querying

Alastair Green
Neo4j
alastair.green@neo4j.com

Martin Junghanns
Neo4j & University of Leipzig
martin.junghanns@neo4j.com

Max Kiessling
Neo4j
max.kiessling@neo4j.com

Tobias Lindaaker
Neo4j
tobias.lindaaker@neo4j.com

Stefan Plantikow
Neo4j
stefan.plantikow@neo4j.com

Petra Selmer
Neo4j
petra.selmer@neo4j.com

ABSTRACT

Cypher is a property graph query language that provides expressive and efficient querying of graph data. Originally designed and implemented within the Neo4j graph database, it is now being used by several industrial database products, as well as open-source and research projects. Since 2015, Cypher has been an open, evolving language, with the aim of becoming a fully-specified standard with many independent implementations.

We introduce Cypher and the property graph model, and then describe extensions – either actively being developed or under discussion – which will be incorporated into Cypher in the near future. These include (i) making Cypher into a fully compositional language by supporting multiple graphs and allowing graphs to be returned from queries; (ii) allowing for more complex patterns (based on regular path queries) to be expressed; and (iii) allowing for different pattern matching semantics – homomorphism, relationship isomorphism (the current default) or node isomorphism – to be configured at a query-by-query level.

A subset of the proposed Cypher language extensions has already been implemented on top of Apache Spark. In the tutorial, we will present our approach including an in-depth analysis of the challenges we faced. This includes mapping the property graph model to the Spark DataFrame abstraction and the translation of Cypher query operators into relational transformations. The tutorial will conclude with a demonstration based on a real-world graph analytical use case.

1 INTRODUCTION

The past few years have seen a marked increase of property graph databases [12] – such as Neo4j [20], Sparksee and JanusGraph – in both the industrial and research arenas. Property graphs have become the model of choice for next-generation graph applications¹. Their use increasingly replaces older approaches to graph data processing such as cross-linked document stores or object-oriented database management systems.

Across both research and industry, property graphs have been used in a wide variety of domains, spanning areas as diverse as fraud detection, recommendations, geospatial data, master data management, network and data centre management, authorisation and access control [23], the analysis of social networks [5], bioinformatics [1, 14, 28] and pharmaceuticals [18], software system analysis [9], and investigative journalism [3].

This trend of increased usage of property graphs is grounded in: (i) their ability to operate on multiple large and highly-connected data sets as one graph that enables novel pattern matching and

graph analytical queries; (ii) their natural ability to cleanly map onto object-oriented or document-centric data models in programming languages; (iii) their visual nature that helps communication between business, application domain, and technical experts; and (iv) their historical development based on the pragmatic needs of real world application developers.

This trend is evidenced by two major factors. The first is the emergence of Cypher as the de-facto standard declarative query language for property graphs, and the second is the growing number of both industrial and academic software products for property graphs.

Since 2015, as part of the openCypher project [22], Cypher has been an open language, and is evolving under the auspices of the openCypher Implementers Group (oCIG), with the aim of becoming a fully-specified standard that can be independently implemented. The recently released Cypher 9 reference [21] along with accompanying formal grammar definitions (EBNF and ANTLR4) and conformance test suite (TCK) – published under the Apache 2.0 license – already provide implementers with a solid basis for adopting Cypher. At the time of writing, Cypher is supported by several commercial systems including SAP HANA Graph [24], Agens Graph, Redis Graph, and Memgraph, along with research frameworks including – in varying degrees of completeness – Gradoop [11], inGraph [15], Cytosm [25], Cypher for Apache Spark [19] and Cypher over Gremlin.

Current developments that are under way include the ability to pass multiple graphs and a table as input to a Cypher query. Moreover, queries will also be able to project and save multiple graphs, and this, coupled with the ability to chain queries together, will render Cypher as the first graph compositional query language. Following on from this work, complex pattern matching and configurable pattern matching semantics will further increase the utility of Cypher in the very near future.

2 SCOPE OF THE TUTORIAL

2.1 Intended audience

This tutorial is aimed at a wide scope of audience, including researchers, students, developers, and industrial practitioners who are interested in the emerging and quickly-evolving area of graph data, databases and languages. All attendees will gain a comprehensive idea of what this field comprises, as well as the future features and challenges that lie ahead for Cypher, the most-used property graph query language.

It is our hope that owing to the many challenges that exist in this area, researchers and students will be motivated to consider this area as a future topic of research.

There are no preliminary requirements for this tutorial, as it will be self-contained and commence with the property graph data model and Cypher, thus assuming no prior knowledge of these.

¹<https://db-engines.com/en/ranking/graph+dbms>

2.2 Goals of the tutorial

The main outcomes of the tutorial comprise:

- A comprehensive understanding of the property graph data model, and how it compares against some of the other graph data models.
- A good understanding of the Cypher property graph query language and its main constructs and features.
- An in-depth treatment of how Cypher will become a fully compositional language through the introduction of multiple graph support and query chaining.
- An overview of Cypher’s version of *regular path queries* in the form of path pattern queries, which additionally include node and relationship property tests to increase the expressivity of Cypher to manage emerging industrial use-cases and requirements.
- An understanding of node and relationship isomorphism and homomorphism, the characteristics of each, the benefits and drawbacks of each (from an industrial point of view) and how these are envisioned to be incorporated into Cypher.
- A good understanding of the Cypher implementation on top of Apache Spark and how to map a schema-free graph data model to a schema-based relational abstraction.
- An understanding of real-world use-cases which can be better solved by using graphs and the proposed language extensions.

3 TUTORIAL OUTLINE

We will begin the tutorial with a brief history of Cypher and the property graph model, and provide an overview of the open-Cypher project and how this is helping to drive forward the design of the language, before proceeding onto the main topics.

3.1 The Cypher property graph query language

Property graph data model. The property graph data model will be described, along with how it originated historically from application use cases. We will compare and contrast property graphs with other graph data models. The tutorial will also contain a discussion of ongoing work on potential extensions to the property graph data model.

Cypher query language. Cypher as it stands today will be presented, focusing on its core elements: pattern matching, path functionality and how updates to the data are performed. We will also cover how Cypher queries are structured, as this will lead into the topic of query composition further on in the tutorial. To set the scene and lay the foundation for the later topics, we will walk through an example query in detail, describing the syntax and semantics at each stage of the query.

Challenge: language evolution. Evolving a language with active users is not a trivial undertaking. Every language change needs to be understood in terms of a plethora of interlocking concerns such as usability, relevant use cases, consistency, ergonomics, syntax, tractability, implementability and performance, as well as aesthetics. Every design decision may have hidden consequences in terms of constraining the design in the future. We will talk about some of these concerns, how we design language changes, and the formal openCypher process for evolving the language.

3.2 Multiple graphs and composition

Multiple graphs. Having set the scene, we will proceed with describing current developments in Cypher. The first of these is the notion of supporting multiple graphs. We will describe how graphs can be referenced, created, updated, saved as a named graph, and projected. We also define a series of set operations on graphs. Throughout, we will use a running example and describe the syntax and semantics at each stage.

Query composition. Having support for multiple graphs, and being able to return one or more graphs from a query paves the way for true graph query composition. Each query can be considered a function, taking as input a table and multiple named graphs, and returning as output a table and multiple named graphs. Thus, a Cypher query can be thought of as a chain of functions, composed of a series of constituent, elementary queries to form a *query chain* or pipeline. The addition of subqueries – a well-known construct from SQL – may be used to transform a query chain into a tree. Named queries – allowing for queries to be re-used in different contexts – will also be presented. We will illustrate these concepts with examples, and show the power and expressivity conferred through graph query composition; we note that, to our knowledge, no other declarative, widely-used query language allows for this.

Challenge: language revolution. Ideal language additions do not interfere with existing features. Sometimes languages need to be changed so substantially that it is impossible to avoid conflicting with pre-existing semantics. In the history of Cypher, various breaking changes have occurred. We will discuss our experiences with breaking changes, language versioning, and planning and executing large scale additions to the language such that the concerns of all relevant stakeholders are incorporated.

3.3 Powerful pattern matching

Path Pattern Queries (PPQs). Regular path queries were first proposed by Cruz, Mendelzon and Wood [4] in 1987, and now, thirty years later, we have turned our attention to this topic and how it may be included in Cypher in the form of *Path Pattern Queries*, or PPQs.

PPQs, inspired by recent work by Libkin, Martens and Vrgoč [13], extend RPQs with notions of node property tests, and are an extremely powerful and expressive mechanism for graph querying. PPQs have been designed to allow for the composition of paths into more complex ones, incorporating both node and relationship property tests, along with the consideration of path costing. We see this as an integral part of Cypher, particularly as the need for users to express ever more complex patterns becomes more pressing in the near future. Using a running example, we will describe the syntax and semantics in detail.

Configurable pattern-matching semantics. The default pattern-matching semantics in Cypher uses relationship (or edge) isomorphism (referred to informally as ‘Cyphermorphism’). Although it has been stated that it is a useful default in most real-world queries [26], there are some cases where a different semantics would be more appropriate. To this end, Cypher will allow the writer of a query to configure the type of pattern-matching semantics the query is to use: either homomorphism, relationship isomorphism, or node isomorphism. We will discuss how this is envisaged to function, and the benefits and drawbacks conferred by each approach.

Challenge: Tractability. Providing more powerful and flexible pattern matching is grounded in ever-growing application requirements. This needs to be balanced against what can be implemented efficiently and what is theoretically tractable. However, in certain cases, these equally valid theoretical and practical viewpoints may be at variance with each other. We will discuss this and provide a perspective on the tensions that exist between theoretical complexity analysis and industrial requirements of graph query languages.

3.4 Cypher for Apache Spark

Graph query languages are currently most prominent in graph database systems such as Neo4j [20]. However, it is our opinion that many systems can benefit from having such a language as part of their feature set. One of these systems is Apache Spark [6], which is one of the most popular open source frameworks in the context of distributed processing of large data volumes within complex analytical workloads.

Apache Spark. Apache Spark is a distributed dataflow framework supporting the declarative definition and execution of distributed dataflow programs sourced from batch data. The basic abstractions of such programs are so-called Resilient Distributed Datasets (RDDs) [29] and transformations between those. A Spark RDD is an immutable, distributed collection of arbitrarily-structured data; transformations are higher-order functions (e.g. map and reduce) that describe the construction of new RDDs either from existing ones or from data sources (e.g., HDFS or RDBMS). To describe an analytical task, a Spark program may include multiple chained transformations. During execution, Spark manages data distribution, parallel execution, load balancing and failover across a cluster of machines.

In addition to the RDD abstraction, Apache Spark includes libraries which offer a higher level of abstraction tailored to specific analytical tasks such as machine learning (SparkML), graph processing (GraphX [8, 27]) and relational operations (SparkSQL). In SparkSQL [2], the abstraction is a so-called DataFrame, which handles structured data according to a fixed schema. Available transformations are well known from relational algebra, comprising, for example, selection, projection, join and grouping. Furthermore, SparkSQL includes Catalyst [2], a rule-based query optimizer that transforms a relational query into an optimized dataflow program by undertaking well-known techniques such as predicate pushdown, column projection and code generation [7].

To incorporate the benefits of Cypher from the graph database domain into the world of distributed dataflow processing, we began developing Cypher for Apache Spark (CAPS) [19]. CAPS is an additional library built on top of SparkSQL and can be integrated into a regular Spark analytical program. CAPS is primarily focused on graph-powered data integration and graph analytical query workloads within the Spark ecosystem. In addition, CAPS is our testbed for Cypher language extensions as specified in the previous sections; for example, query composition, graph transformation and multiple graphs.

Challenge: Schema-flexible mapping. In order to benefit from the query optimization capabilities of Catalyst, we decided to implement CAPS on top of SparkSQL.² This however introduces the problem of mapping the schema-flexible property graph model to

a schema-fixed DataFrame representation. We solve this problem by defining a graph schema, which includes information about node labels, relationship types and associated properties potentially having conflicting data types. For structured data sources, such as CSV files or RDBMSs, the schema can be derived directly from meta data supplied by the data source. However, unstructured or semi-structured data sources – exemplified by native graph databases such as Neo4j or document databases such as CosmosDB [17] – will require a full scan of the source data to compute the schema, should it not exist in the first instance. Once a schema is available, it is used to split node and relationship data into multiple column entries, i.e., a row inside a structured DataFrame (“flatten out nodes and relationships”) resulting in potentially sparse tables. In the tutorial, we will discuss the process of schema computation and node / relationship flattening in detail including more information about our type system.

Challenge: Multi-phase planning. A second challenge we faced when building CAPS was the translation of a Cypher graph query to a sequence of relational operations on the Spark DataFrame API. Our implementation approach is based on our experiences from building the Neo4j query planner as well as several existing publications discussing the formal aspects of that topic [10, 11, 16]. CAPS uses multiple compilation phases to produce an executable Spark program, including: building a canonical query representation from an abstract syntax tree; translating the canonical form into graph-specific query operators (logical planning); computing the schema for intermediate results (flat planning); and translating logical operators into Spark DataFrame transformations (physical planning). A physical CAPS plan is optimized by Catalyst and translated into an executable Spark program. In the tutorial, we give an introduction into the CAPS compilation phases as well as optimization techniques like reuse of intermediate results, tree rewriting and Catalyst optimization.

3.5 Demonstration

To highlight the analytical benefits of the graph data model as well as the Cypher query language and its proposed extensions, we will end the tutorial with a live demonstration of CAPS. The demonstration will illustrate a hypothetical analytical workflow including graph data integration from multiple data sources, graph transformation and graph analytical queries. We also demonstrate the integration of CAPS within the Spark ecosystem by using Apache Zeppelin, a tool for browser-based interactive data analytics.

4 PRESENTER BIOGRAPHIES

Alastair Green is a Director of Product Management at Neo4j and is a member of the openCypher Language Group, supporting the openCypher project.

Martin Junghanns is part of the Cypher for Apache Spark Engineering team at Neo4j. Apart from that, he is finishing his PhD in Computer Science at the University of Leipzig. Martin is working on the GRADOOP project with a focus on distributed graph analytics, graph data models and analytical DSLs.

Max Kiessling is part of the Cypher for Apache Spark Engineering team at Neo4j. He recently finished his Master’s thesis at the University of Leipzig, in which he researched distributed pattern matching as part of the GRADOOP project.

Tobias Lindaaker is one of the first engineers to have worked at Neo4j. He has been and continues to be a key influencer in

²The CAPS architecture is backend-agnostic and can be ported to alternative backends / systems.

the evolution of the property graph model, as well as the Cypher query language.

Stefan Plantikow is a member of the Engineering team at Neo4j and is leading the openCypher Language Group. He has worked on the Neo4j kernel, the original implementation of Cypher and the current Neo4j query planner, as well as the BOLT protocol. Stefan has a Master's degree in Computer Science from Humboldt University, Berlin, in the area of distributed systems.

Petra Selmer is a member of Engineering team at Neo4j and is a member of the openCypher Language Group, supporting the openCypher project. For many years, she worked as a consultant and developer in a variety of different domains and roles and has a PhD in Computer Science from Birkbeck, University of London, where she researched the flexible querying of graph-structured data.

REFERENCES

- [1] Davide Alloci, Julien Mariethoz, Oliver Horlacher, Jerven T. Bolleman, Matthew P. Campbell, and Frederique Lisacek. 2015. Property Graph vs RDF Triple Store: A Comparison on Glycan Substructure Search. *PLoS ONE* 10, 12 (12 2015), 1–17.
- [2] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*.
- [3] Mar Cabra. 2016. How the ICIJ used Neo4j to unravel the Panama Papers. Neo4j Blog. (May 2016). <https://neo4j.com/blog/icij-neo4j-unravel-panama-papers/>.
- [4] Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. 1987. A Graphical Query Language Supporting Recursion. In *SIGMOD Conference*. ACM Press, 323–330.
- [5] Georgios Drakopoulos, Andreas Kanavos, and Athanasios K. Tsakalidis. 2016. Evaluating Twitter Influence Ranking with System Theory. In *Proceedings of the 12th International Conference on Web Information Systems and Technologies, WEBIST 2016, Volume 1, Rome, Italy, April 23-25, 2016*. 113–120.
- [6] Apache Software Foundation. 2017. Apache Spark. <https://spark.apache.org/>. (accessed: Dec. 2017).
- [7] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. 2008. *Database Systems: The Complete Book*.
- [8] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*.
- [9] Nathan Hawes, Ben Barham, and Cristina Cifuentes. 2015. FrappÉ: Querying the Linux Kernel Dependency Graph. In *Proceedings of the GRADES'15 (GRADES'15)*. ACM, 4:1–4:6.
- [10] Jürgen Hölsch and Michael Grossniklaus. 2016. An Algebra and Equivalences to Transform Graph Patterns in Neo4j. In *Proceedings of the Workshops of the EDBT/ICDT 2016 Joint Conference, EDBT/ICDT Workshops 2016, Bordeaux, France, March 15, 2016*.
- [11] Martin Junghanns, Max Kießling, Alex Averbuch, André Petermann, and Erhard Rahm. 2017. Cypher-based Graph Pattern Matching in Gradoop. In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems (GRADES '17)*.
- [12] Josep-Lluís Larriba-Pey, Norbert Martínez-Bazan, and David Domínguez-Sal. 2014. Introduction to Graph Databases. In *Reasoning Web (Lecture Notes in Computer Science)*, Vol. 8714. Springer, 171–194.
- [13] Leonid Libkin, Wim Martens, and Domagoj Vrgoč. 2016. Querying graphs with data. *Journal of the ACM* 63, 2 (2016), 14:1–14:53.
- [14] Artem Lysenko, Irina A. Roznovat, Mansoor Saqi, Alexander Mazein, Christopher J. Rawlings, and Charles Auffray. 2016. Representing and querying disease networks using graph databases. *BioData Mining* 9, 1 (25 Jul 2016), 23.
- [15] József Marton, Gábor Szárnyas, and Márton Búr. 2017. Model-Driven Engineering of an OpenCypher Engine: Using Graph Queries to Compile Graph Queries. In *SDL Forum (Lecture Notes in Computer Science)*, Vol. 10567. Springer, 80–98.
- [16] József Marton, Gábor Szárnyas, and Dániel Varró. 2017. *Formalising open-Cypher Graph Queries in Relational Algebra*.
- [17] Microsoft. 2017. CosmosDB. <https://docs.microsoft.com/en-us/azure/cosmos-db/introduction>. (accessed: Dec. 2017).
- [18] Joseph Mullen, Simon J. Cockell, Peter Woollard, and Anil Wipat. 2016. An Integrated Data Driven Approach to Drug Repositioning Using Gene-Disease Associations. *PLoS ONE* 11, 5 (05 2016), 1–24.
- [19] Neo4j. 2017. Cypher-for-Apache-Spark. <https://github.com/opencypher/cypher-for-apache-spark>. (accessed: Dec. 2017).
- [20] Neo4j. 2017. Neo4j Database. <http://www.neo4j.com/>. (accessed: Dec. 2017).
- [21] openCypher. 2017. Cypher Query Lang. Ref. <https://github.com/opencypher/opencypher/blob/master/docs/openCypher9.pdf>. (accessed: Dec. 2017).
- [22] openCypher. 2017. openCypher. <http://www.opencypher.org/>. (accessed: Dec. 2017).
- [23] Ian Robinson, Jim Webber, and Emil Eifrem. 2013. *Graph databases*. O'Reilly Media.
- [24] Michael Rudolf, Marcus Paradies, Christof Bornhövd, and Wolfgang Lehner. 2013. The Graph Story of the SAP HANA Database. In *Datenbanksysteme für Business, Technologie und Web (BTW), 15. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 11.-15.3.2013 in Magdeburg, Germany. Proceedings*. 403–420.
- [25] Benjamin A. Steer, Alhamza Alnaimi, Marco A. B. F. G. Lotz, Felix Cuadrado, Luis M. Vaquero, and Joan Varvenne. 2017. Cytosm: Declarative Property Graph Queries Without Data Migration. In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems (GRADES'17)*. 4:1–4:6.
- [26] Oskar van Rest. 2017. Graph pattern matching semantics. <https://tinyurl.com/oCIM1-patternmatching>. (accessed: Dec. 2017).
- [27] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. 2013. GraphX: A Resilient Distributed Graph System on Spark. In *First International Workshop on Graph Data Management Experiences and Systems (GRADES '13)*.
- [28] Byoung-Ha Yoon, Seon-Kyu Kim, and Seon-Young Kim. 2017. Use of Graph Database for the Integration of Heterogeneous Biological Data. *Genomics & informatics* (03 2017), 19–27.
- [29] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*.

Real-Time Data Management for Big Data

Extended Abstract

Wolfram Wingerath
 University of Hamburg
 Hamburg, Germany
 wingerath@informatik.uni-hamburg.de

Felix Gessert
 Baqend GmbH
 Hamburg, Germany
 fg@baqend.com

Erik Witt
 Baqend GmbH
 Hamburg, Germany
 ew@baqend.com

Steffen Friedrich
 University of Hamburg
 Hamburg, Germany
 friedrich@informatik.uni-hamburg.de

Norbert Ritter
 University of Hamburg
 Hamburg, Germany
 ritter@informatik.uni-hamburg.de

ABSTRACT

Users have come to expect reactivity from mobile and web applications, i.e. they assume that changes made by other users become visible immediately. However, developers are challenged with building reactive applications on top of traditional pull-oriented databases, because they are ill-equipped to push new information to the client. Systems for data stream management and processing, on the other hand, are natively push-oriented and thus facilitate reactive behavior, but they do not follow the same collection-based semantics as traditional databases: Instead of database collections, stream-oriented systems are based on a notion of potentially unbounded sequences of data items.

In this tutorial, we survey and categorize the system space between pull-oriented databases and push-oriented stream management systems, using their respectively facilitated means of data retrieval as a reference point. A particular emphasis lies on the novel system class of real-time databases which combine the push-based access paradigm of stream-oriented systems with the collection-based query semantics of traditional databases. We explore why real-time databases deserve distinction in a separate system class and dissect their different architectures to highlight issues, derive open challenges, and discuss avenues for addressing them.

1 INTRODUCTION

Reactive applications require the underlying data storage to publish new and updated information as soon as it is created; data access is *push-based*. In contrast, traditional **database management** systems [6] have been tailored towards *pull-based* data access where information is only made available as a direct response to a client request. While triggers and other push-oriented mechanisms have been added to their initial design, they are outperformed by several orders of magnitude when held against natively push-based systems [9]. In consequence, the inadequacy of traditional database technology for handling rapidly changing data has been widely accepted as one of the fundamental challenges in database design [8].

To warrant low-latency updates in quickly evolving domains, **data stream management** systems [5] break with the idea of

maintaining a persistent data repository. Instead of random access queries on static collections, they perform sequential, long-running queries over data streams. Data stream management systems generate new output whenever new data becomes available and are thus natively push-based. However, data is only available for processing in one single pass, because data streams are conceptually *unbounded* sequences of data items and therefore infeasible to retain indefinitely. Consequently, queries over streams are confined to data that arrives after query activation.

Database Management	Data Stream Management
pull-based	push-based
persistent collection	ephemeral stream
ad hoc, random access	continuous, sequential

Table 1: A side-by-side comparison of core characteristics of database and data stream management systems.

Database and data stream management, respectively, follow fundamentally different semantics regarding the way that data is processed and accessed as Table 1 summarizes. The concept of **persistent collections** conforms to applications that require a (consistent) view of their domain, for instance to keep track of warehouse stock or do financial accounting. The **data stream** model, on the other hand, comes natural for domains that entertain a notion of event sequences or need to reason about the relationship between events, for example to analyze stock prices or identify malicious user behavior. However, the access paradigm – pull-based or push-based – is tied to the data model: Database management systems lack support for **continuous queries** over collections, whereas data stream management systems only provide limited options for persistent data handling.

Acknowledging the gap between traditional databases on the one side and data stream management and stream processing systems on the other, a new class of information systems has emerged that combines collection-based semantics with a push-based access model. These systems are often referred to as **real-time databases** [10, 13], because they keep data at the client in-sync with current database state “in realtime” [7], i.e. as soon as possible after change. Like traditional databases, they store consistent snapshots of domain knowledge. But like stream management systems, they allow clients to subscribe to long-running queries that push incremental updates.

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

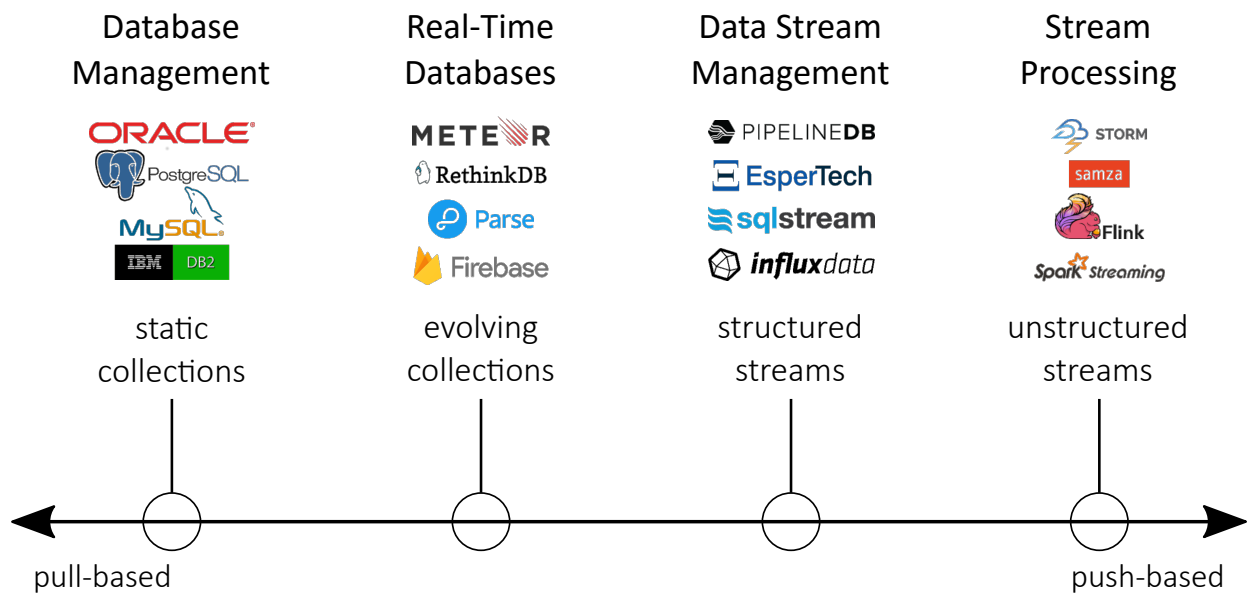


Figure 1: Different classes of data management systems and the access patterns they support.

2 SYSTEM LANDSCAPE OVERVIEW: PULL VS. PUSH

We think systems for data management can be classified by the way they facilitate access to data as illustrated in Figure 1. At the one extreme, there are **traditional databases** which represent snapshots of domain knowledge as the basis of all queries. At the other extreme, there are general-purpose **stream processing** engines which are designed to generate output from conceptually unbounded and arbitrarily structured ephemeral data streams. Real-time databases and data stream management systems both stand in the middle, but adhere to different semantics: **Real-time databases** work on evolving collections that are distinguished from their static counterparts (i.e. from database collections) through continuous integration of updates over time, enabling continuous (real-time) queries over database collections. **Data stream management** systems, as the name implies, provide APIs to query data streams, for example by filtering specific data and computing rolling aggregations and joins over to-be-specified time windows. In contrast to most general-purpose stream processors, datastream management systems usually support pull-based access to some degree, e.g. in the form of common ad hoc database queries over the set of currently retained records.

2.1 Static Queries vs. Continuous Queries

A *pull-based* (static) query assembles data from a bounded data repository and completes by returning data once, whereas a *push-based* (continuous) query processes a conceptually unbounded stream of information to generate incremental output over time. Given these fundamental differences, the design of any data management system reflects a bias towards one or the other; for example, while databases support push-based data access to a certain degree (e.g. through triggers), they are clearly geared towards efficiency for pull-based data retrieval.

2.2 Collections vs. Streams

While a database collection represents the current *state* of the application domain, a data stream rather encapsulates recent *change*.

A **stream**-based representation of an application domain provides a sequential view on events as they occur, but does not retain them indefinitely: Data items are available for a certain time window and are discarded eventually. This view on the data promotes use cases that require notifications, but queries do not reflect actions that happened long ago, since the system only operates on a suffix and not the entirety of event history. In order to serve historical data, the ephemeral events have to be applied to a persistent representation of application state.

A database **collection** reflects all data ever written and thus enables queries that take all events into account. Since collection-based ad hoc queries only generate one single output, though, traditional databases do not propagate informational updates to the client.

2.3 Real-Time Queries Over Database Collections

Given a database's limitation to mainly pull-based access, reactive user interfaces are hard to build on top of an ordinary database. One possibility is to reevaluate a given collection-based query from time to time which is inefficient and introduces staleness on the order of the refresh interval. Another approach is to merge results from collection-based and stream-based queries; thus, the application is effectively burdened with the task of view maintenance which is complex and error-prone. Real-time databases aim to close the gap between both paradigms by providing collection-based semantics for pull-based and push-based queries alike.

3 IN-DEPTH SURVEY: REAL-TIME DATABASES

Our real-time database survey will concentrate on the systems we perceive as the most popular. Due to space limitations, we do

	Meteor		RethinkDB	Parse	Firestore
	poll-and-diff	oplog tailing			
scales with write throughput	✓	✗	✗	✗	?
scales with number of queries	✗	✓	✓	✓	?
composite queries (AND/OR)	✓	✓	✓	✓	✗
sorted queries	✓	✓	✓	✗	○ <small>(single attribute)</small>
limit	✓	✓	✓	✗	✓
offset	✓	✓	✗	✗	✓
aggregations	✗	✗	✗	✗	✗
joins	✗	✗	✗	✗	✗
event stream queries	✓	✓	✓	✓	✓
self-maintaining queries	✓	✓	✗	✗	✗

Table 2: A direct comparison of the different real-time query implementations covered in the in-depth survey.

not discuss these systems in the extended tutorial abstract and refer to our written survey for reference [11]. Table 2 sums up the respective capabilities of each system detailed in our discussion: Meteor, RethinkDB and Parse provide complex real-time queries, but present scale-prohibitive bottlenecks in their respective architectures. While the technology stack behind Firestore is not disclosed, it is apparent that Firestore avoids scalability issues by simply not offering complex queries to begin with.

3.1 Open Challenges

In concept, real-time databases extend traditional databases as they follow the same semantics, but provide an additional mode of access. In practice, though, there is no established scheme how to build a practically useful real-time database system. As will be shown in the tutorial, every push-based real-time query mechanism is deficient in at least one of the following characteristics:

- (1) **scalability:** Serving real-time queries is a resource-intensive process which requires continuous monitoring of all write operations that might possibly affect query results. To sustain more demanding workloads than a single machine could handle, real-time databases typically partition the set of queries across database nodes. As each node is only responsible for a subset of all queries in this scheme, most systems can scale with the number of concurrent queries. However, we are not aware of any real-time database that supports partitioning the change stream as well. Thus, responsibility for individual queries is not shared among nodes and overall system throughput remains bottlenecked by single-machine capacity: Queries simply become intractable as soon as one node is not able to keep up with processing the entire change stream.
- (2) **expressiveness:** The majority of real-time query APIs are limited in comparison to their ad hoc counterparts. Aggregations are generally not available and sorting queries are

often unsupported or have severe restrictions; for example, there are implementations that only allow ordering by a single attribute or offer a limit, but no offset clause. The lack of such basic functionality on the database side necessitates inefficient workarounds in the application code, even for moderately sophisticated data access patterns.

- (3) **legacy support:** Today’s real-time databases have been designed from scratch or on top of NoSQL datastores [11] that do not follow standards regarding data model or query language. They implement custom protocols for pull-based and push-based data access alike and exhibit interfaces that are incompatible among different vendors. While the complete lack of support for legacy interfaces (particularly SQL) may be acceptable in development of a new application, it complicates the adoption of push-based queries in the context of existing technology stacks.
- (4) **abstract API:** Many real-time query APIs expose specificities of the underlying implementation and thus offer poor data independence. As such, these interfaces reflect bottom-up design and force developers to reason about problems that lie beyond the application domain. For example, most real-time databases do not provide interfaces that can be used without knowledge of system internals. Instead, they mostly require an understanding of internal mechanisms or the structure of change events.

During the talk, we will illustrate how the above-mentioned limitations present themselves in practice. We also identify the underlying issues in the respective system architectures and discuss possibilities to avoid them in future designs. In this context, we will discuss related technology (e.g. distributed stream processing engines [12]) and use them as a source of inspiration for resolving the apparent challenges.

4 DIFFERENTIATION FROM OTHER VERSIONS OF THE TUTORIAL

The survey of stream processing engines and the overview over real-time databases have already been presented at different occasions, e.g. at BTW 2017 [4]. Some of the use cases that will be presented have been discussed in our VLDB 2017 industry paper [2]. However, since two of the authors (Wolfram Wingerath and Felix Gessert) are just now finishing their Ph.D. theses on real-time big data management, the tutorial intended for March 2018 will incorporate significant updates and extensions. In particular, the scientific portion of the talk will be amended by recent developments in the space of real-time databases. Further, we will present our experiences in building and using a real-time database in customer-facing applications at the Backend-as-a-Service company Baqend. Thus, the tutorial will provide a unique combination of broad scientific research and real-world experiences.

5 SCOPE, LENGTH & INTENDED AUDIENCE

The tutorial in the form outlined here is intended for 90 minutes and will concentrate on push-based systems, namely real-time databases and stream processing engines. We can also extend this tutorial to 180 minutes by including our previous tutorials on NoSQL database systems [1, 3, 4] and discussing them in the light of real-time and stream processing requirements. This tutorial is intended for anybody interested in novel database technology; there are no prerequisites, even though a certain technical understanding of databases will be helpful in following the in-depth discussion.

6 PRESENTER BIOGRAPHIES

Wolfram Wingerath is a Ph.D. student under supervision of Norbert Ritter teaching and researching at the University of Hamburg. He was co-organiser of the BTW 2015 conference and has held workshop and conference talks on his published work on several occasions. Wolfram is part of the databases and information systems group and his research interests evolve around real-time databases and related technology such as scalable stream processing, NoSQL database systems, cloud computing, and Big Data analytics. His Ph.D. thesis explores a scalable design for push-based real-time queries on top of pull-based databases.

Felix Gessert is a Ph.D. student at the databases and information systems group at the University of Hamburg. His main research fields are scalable database systems, transactions, and web technologies for cloud data management. His thesis addresses caching and transaction processing for low-latency mobile and web applications. He is also founder and CEO of the startup Baqend that implements these research results in a cloud-based backend-as-a-service platform. Since their product is based on a polyglot, NoSQL-centric storage model, he is very interested in both the research and practical challenges of leveraging and improving these systems. He is frequently giving talks on different NoSQL topics.

Erik Witt is a Full Stack developer and performance engineer at Baqend where he builds and optimizes scalable web applications for the cloud. As the highlight of his master's degree at the university and in cooperation with Baqend, he developed a web-caching-based transaction concept for distributed cloud databases. Erik has talked about his work at numerous conferences and also regularly authors articles on the Baqend company blog and related media in order to present the intricacies of web performance to a broader audience.

Steffen Friedrich is a Ph.D. student working under supervision of Norbert Ritter at the University of Hamburg. He has taken part in several workshops and conferences, both as presenter and as co-organiser (BTW 2015). Being a member of the databases and information systems group, Steffen is interested in large-scale data management and data-intensive computing. Furthermore, in his Master thesis, he also dealt with data quality issues, specifically with duplicate detection in probabilistic data. His research project is primarily concerned with benchmarking of non-functional characteristics (e.g. consistency and availability) in distributed NoSQL database systems.

Norbert Ritter is a full professor of computer science at the University of Hamburg, where he heads the databases and information systems group. He received his Ph.D. from the University of Kaiserslautern in 1997. His research interests include distributed and federated database systems, transaction processing, caching, cloud data management, information integration, and autonomous database systems. He has been teaching NoSQL topics in various courses for several years. Seeing the many open challenges for NoSQL systems, he and Felix Gessert have been organizing the annual Scalable Cloud Data Management Workshop (www.scdm.cloud) to promote research in this area.

REFERENCES

- [1] Felix Gessert and Norbert Ritter. 2016. Scalable Data Management: NoSQL Data Stores in Research and Practice. In *32nd IEEE International Conference on Data Engineering, ICDE 2016*.
- [2] Felix Gessert, Michael Schaarschmidt, Wolfram Wingerath, Erik Witt, Eiko Yoneki, and Norbert Ritter. 2017. Quaestor: Query Web Caching for Database-as-a-Service Providers. *Proceedings of the 43rd International Conference on Very Large Data Bases (2017)*, 12.
- [3] Felix Gessert, Wolfram Wingerath, Steffen Friedrich, and Norbert Ritter. 2016. NoSQL Database Systems: A Survey and Decision Guidance. *Computer Science - Research and Development (2016)*.
- [4] Felix Gessert, Wolfram Wingerath, and Norbert Ritter. 2017. Scalable Data Management: An In-Depth Tutorial on NoSQL Data Stores. In *Datenbanksysteme für Business, Technologie und Web (BTW 2017) - Workshopband, 2.-3. März 2017, Stuttgart, Germany*.
- [5] Lukasz Golab and M. Tamer Zsu. 2010. *Data Stream Management*. Morgan & Claypool Publishers.
- [6] Joseph M. Hellerstein, Michael Stonebraker, and James Hamilton. 2007. Architecture of a Database System. *Found. Trends databases 1*, 2 (Feb. 2007), 141–259. <https://doi.org/10.1561/1900000002>
- [7] Ryan Paul. 2015. Build a realtime liveblog with RethinkDB and PubNub. *RethinkDB Blog* (May 2015). <https://rethinkdb.com/blog/rethinkdb-pubnub/> Access: 2017-05-20.
- [8] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. 2005. The 8 Requirements of Real-time Stream Processing. *SIGMOD Rec.* 34, 4 (Dec. 2005), 42–47. <https://doi.org/10.1145/1107499.1107504>
- [9] Michael Stonebraker and Uğur Cetintemel. 2005. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *Proceedings of the 21st International Conference on Data Engineering (ICDE '05)*. IEEE Computer Society, Washington, DC, USA, 2–11.
- [10] Frank van Puffelen. 2016. Have you met the Realtime Database? *The Firebase Blog* (July 2016). <https://firebase.googleblog.com/2016/07/have-you-met-realtime-database.html> Accessed: 2017-05-20.
- [11] Wolfram Wingerath. 2017. Real-Time Databases Explained: Why Meteor, RethinkDB, Parse and Firebase Don't Scale. *Baqend Tech Blog* (2017). <https://medium.com/p/822ff87d2f87>
- [12] Wolfram Wingerath, Felix Gessert, Steffen Friedrich, and Norbert Ritter. 2016. Real-time stream processing for Big Data. *it - Information Technology* 58, 4 (2016), 186–194. <https://doi.org/10.1515/itit-2016-0002>
- [13] Alice Yu. 2015. What does it mean to be a real-time database? – Slava Kim at Devshop SF May 2015. *Meteor Blog* (June 2015). Accessed: 2017-05-20.

Supporting Similarity Queries in Apache AsterixDB

Taewoo Kim¹ Wenhai Li² Alexander Behm[†] Inci Cetindil[†] Rares Vernica[†]
 Vinayak Borkar[†] Michael J. Carey¹ Chen Li¹

¹University of California, Irvine, CA, USA ²Wuhan University, China

ABSTRACT

Many applications require similarity query processing. Most existing work took an algorithmic approach, developing indexing structures, algorithms, and/or various optimizations. In this work, we choose to take a different, systems-oriented approach. We describe the support for similarity queries in Apache AsterixDB, a parallel, open-source Big Data management system for NoSQL data. We describe the lifecycle of a similarity query in the system, including the support provided at the query language level, indexing, execution plans (with and without indexes), plan rewrites to optimize query execution, and so on. Our approach leverages the existing infrastructure of AsterixDB, including its operators, parallel query engine, and rule-based query optimizer. We have conducted an experimental study using several large, real data sets on a parallel computing cluster to evaluate AsterixDB's support for similarity queries, and we share the efficacy and performance results here.

1 INTRODUCTION

Similarity queries compute answers satisfying matching conditions that are not exact but approximate. The problem of supporting similarity queries has become increasingly important in many applications, including search, record linkage [1], data cleaning [27], and social media analysis [4]. For instance, during a live phone conversation with a client, a call center representative might wish to immediately identify a product purchased by the customer by typing in a serial number. The system should locate the product even in the presence of typos in the search number. A social media analyst might want to find user accounts that share common hobbies or social friends. A medical researcher may want to search for papers whose title is similar to a particular article. In each of these examples the query includes a matching condition with a similarity function that is domain specific, such as edit distance for a keyword or Jaccard for sets of hobbies.

There are two basic types of similarity queries. One is *search*, or *selection*, which finds records similar to a given record. The other is *join*, which computes pairs of records that are similar to each other. There have been many studies on these two types of queries, both with and without indexes. A plethora of data structures, partitioning schemes, and algorithms have been developed to support similarity queries efficiently on large data sets. When the computation is beyond the limit of a single computer, there are also parallel solutions that support queries across multiple nodes in a cluster. (See Section 1.1 for an overview.) The techniques developed in the last two decades have significantly improved the performance of similarity queries and have enabled applications to support such queries on millions or even billions of records.

Most existing work has taken an algorithmic approach, developing index structures and/or algorithmic optimizations. We

have taken a different, systems-oriented approach – tackling the problem of supporting similarity queries *end-to-end* in a full, declarative parallel data management system setting. Here we explain how such queries are supported in Apache AsterixDB, an open-source parallel data management system for semi-structured (NoSQL) data. By “end-to-end”, we refer to the whole lifecycle of a query, including query language support for similarity conditions, internal index structures, execution plans with or without an index, plan rewriting to optimize execution, and so on.

Achieving our goal has involved several challenges. First, as similarity in queries can be domain specific, we need to support commonly used functions as well as letting users provide their own customized functions. Second, due to the complex logic of existing algorithms, we need to consider how to support them using existing operators without “reinventing the wheel” (without introducing new, ad hoc operators). Third, we need to consider how to leverage an existing query optimizer framework to rewrite similarity queries to achieve high performance. In this paper we discuss these challenges and offer the following contributions:

(1) We show how to extend the existing query language of AsterixDB to allow users to specify a similarity query, either by using a system-provided function or specifying their own logic as a user-defined function (Section 3).

(2) We show how to implement state-of-the-art techniques using existing operators in AsterixDB, both for index-based and non-index-based plans (Section 4) and for both search and join queries. Our solution not only allows the query plans to benefit from the built-in optimizations in those operators, but also to automatically enjoy future improvements in these operators.

(3) We show how to rewrite similarity queries in an existing rule-based optimization framework (Section 5). A plan for an ad hoc similarity join can be very complex. As an example, a three-stage join plan based on the technique in [34] can involve up to 77 operators (Section 5.2). To enable the optimizer to more easily transform such complex plans, we developed a novel framework called “AQL+” that takes a two-step approach to rewriting a plan. A major advantage of the framework is that it allows AsterixDB to support queries with more than one similarity join condition, making it the first parallel data management system (to our knowledge) to support similarity queries with multiple similarity joins.

(4) We present an empirical study using several large, real data sets on a parallel cluster to evaluate the effects of these techniques. The results show the efficacy of AsterixDB's support for parallel similarity queries on large data sets. (Section 6).

1.1 Related Work

There are various kinds of similarity queries on strings and sets. For string similarity search, many algorithms use a gram-based approach (e.g., [5, 20, 29]). VGRAM [19] extends the approach by introducing variable-length grams. For string similarity join, filtering techniques are widely used. Length filtering uses the

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

²Contact author and part of his work was done when visiting UC Irvine. [†]Work done when affiliated with UC Irvine.

length of a string to reduce the number of candidates. For example, an algorithm called gram-count [15] utilizes the fact that for two strings to be similar based on a threshold δ , their length difference should be within δ . Prefix filtering [3, 7, 12, 22, 26, 28, 35, 37–39] utilizes the fact that two strings are similar only if they share some commonality in their prefixes. Many algorithms have been proposed based on this observation, such as All-Pair [3], PPJoin [39], PPJoin+ [39], MPJoin [28], ED-Join [38], AdaptJoin [35], QChunk [26], VChunk [37], and Pivotal prefix [12]. Other related algorithms exist such as M-Tree [9] and trie-Join [14]. There have been several evaluation studies about string-similarity [18] and set-similarity joins [24]. There is a recent survey about string similarity queries [25]. The authors of [18] found that AdaptJoin [35] and PPJoin+ [39] were best for Jaccard similarity. Meanwhile, the authors of [24] concluded that AllPair [3] was still competitive. The authors of [25] discussed prefix-filtering techniques. Many of these algorithms assume the data to be searched or joined can fit in main memory.

For parallel similarity join, a number of studies have used the MapReduce framework [11, 21, 31, 34, 36]. There is one survey that discusses parallel similarity join [13]. Vernica et al. [34] proposed a three-stage algorithm in such a setting. There are also studies on integrating similarity join into database management systems [10, 15, 16, 30, 32]. Some adopted similarity join as a UDF or express a similarity join in a SQL expression; others introduced a relational operator to support similarity joins.

Our focus is different, as it is about supporting similarity in a general-purpose parallel database system context. We needed to address various systems-oriented challenges when adopting existing techniques in this context. System-wise, a parallel similarity query processing system, Dima [33], has been proposed recently. A key difference is that Dima is an in-memory based system, unlike AsterixDB. There are some search systems and DBMSs that support similarity queries, including Elasticsearch, Oracle, and Couchbase. Unlike AsterixDB, Elasticsearch is middleware and it focuses on search, not join. Oracle supports edit distance via an extension package if a specific type of index is created. Couchbase supports edit distance searches on NoSQL data in its new full-text search service, but only via a separate full-text API (not its N1QL query language). In contrast, AsterixDB provides a general class of similarity functions for strings that work for both select and join operations, and a similarity predicate can be part of a general query along with non-similarity predicates.

2 PRELIMINARIES

2.1 Similarity Functions

A similarity measure is used to represent the degree of similarity between two objects. An object can be a string or a bag of elements. There are various types of similarity measures depending on the objects that are being compared. In this paper, we focus on two widely used classes of measures, namely string-similarity functions and set-similarity functions.

String-Similarity Functions: One widely used string similarity function is edit distance, also known as Levenshtein distance. The edit distance between two strings r and s is the minimum number of single-character operations (insertion, deletion, and substitution) required to transform r to s . For instance, the edit distance between “james” and “jamie” is 2, because the former can be transformed to the latter by inserting “i” after “m” and deleting “s”. There are other string-similarity functions such as Hamming distance and Jaro-winkler distance.

Set-Similarity Functions: These are used to represent the similarity between two sets. There are many such functions, such as Jaccard, dice, and cosine. In this paper, we focus on Jaccard similarity, which is one of the most common set-similarity measures. For two sets r and s , their Jaccard similarity is $Jaccard(r, s) = \frac{|r \cap s|}{|r \cup s|}$. For example, the Jaccard similarity between $r = \{\text{“Good”, “Product”, “Value”}\}$ and $s = \{\text{“Nice”, “Product”}\}$ is $\frac{1}{4}$. Such set-similarity functions can also be utilized to measure the similarity between two strings by tokenizing them (i.e., into n -grams or words) and measuring the set similarity of their token multisets. Dice and cosine values can be calculated similarly.

Similarity Search: Similarity search finds all objects in a collection that are similar to a given object based on a given similarity metric. Let sim be a similarity function, and δ be a similarity threshold. An object r from a collection R is similar to a query object q if $sim(r, q) \geq \delta$.

Similarity Join: Joins find similar $\langle r, s \rangle$ pairs of objects from two collections R and S , where $r \in R$, $s \in S$, and $sim(r, s) \geq \delta$.

2.2 Answering Similarity Queries

For similarity queries, using a brute-force, scan-based algorithm is computationally expensive, so there have been many studies in the literature to support similarity queries more efficiently. One widely used method is the gram-based approach, which utilizes the n -grams of a string. An n -gram of a string r is a substring of r with length n . For instance, the 2-grams of string “james” are {“ja”, “am”, “me”, “es”}.

review-id	username	review-summary
1	james	This movie touched my heart!
2	mary	The best car charger I ever
3	mario	Different than my usual but good
4	jamie	Great Product - Fantastic Gift
5	maria	Better ever than I expected

Figure 1: Example data of Amazon reviews (simplified).

String-similarity queries can be answered by utilizing an n -gram inverted index. For each gram g of the strings in a collection R , there is an inverted list l_g of the ids of the strings that include this gram g . Figure 2 shows the inverted lists for the 2-grams of the “username” field of the sample Amazon reviews in Figure 1.

gram	am	ar	es	ia	ie	io	ja	ma	me	mi	ri	ry
inverted list	1 4	2 3 5	1	5	4	3	1 4	2 3 5	1	4	3 5	2

Figure 2: Inverted lists for 2-grams of the “username” field.

We can answer a string-similarity query by computing the n -grams of the query string and retrieving the inverted lists of these grams. We then process the inverted lists to find all string ids that occur at least T times, since a string r within edit distance k of another string s must share at least $T = |G(r)| - k \times n$ grams with s [17]. This problem is called the T -occurrence problem. Solving the T -occurrence problem yields a set of candidate string ids. The false positives are removed in a final verification step by fetching the strings of the candidate string ids and computing their real similarities to the query. As an example, given a gram length $n = 2$, an edit distance threshold $k = 1$, and a query string $q = \text{“marla”}$, Figure 3 illustrates how to find the similar usernames from the data in Figure 1. We first compute the 2-grams of q as {“ma”, “ar”, “rl”, “la”} and retrieve the inverted lists of these 2-grams. We consider the records that appear at least $T = 4 - 2 \times 1 = 2$ times on these lists as candidates, which have review-ids 2, 3, and 5. Last, we compute the real similarity for these candidates, and

the review-id 5 is the final answer. Note that if the threshold $T \leq 0$, then the entire data collection needs to be scanned to compute the results, which is called a *corner case*. In the above example, if the threshold is 3, then $T = 4 - 2 \times 3 = -2$, causing a corner case.

ma	ar	rl	la	Candidate	Verification
2	2	-	-	2	x
3	3			3	x
5	5			5	✓

Figure 3: Answering an edit-distance query for “q”=marla and $T=2$.

2.3 Apache AsterixDB

Initiated in 2009, the AsterixDB project integrated ideas from three distinct areas – semi-structured data, parallel databases, and data-intensive computing – to create an open-source software platform that scales on large, shared-nothing commodity computing clusters. AsterixDB consists of several software layers. The top-most layer provides a parallel DBMS with a full, flexible data model (ADM) and query languages (AQL/SQL++) for describing, querying, and analyzing data. The next layer, a query compiler based on *Algebricks* [8], is used for parallel query processing. This algebraic layer receives a translated query plan from the upper layer and transforms it using rule-based optimization. It also generates Hyracks jobs to be executed on the *Hyracks* [6] layer. It provides storage facilities for datasets that are stored and managed by AsterixDB as partitioned LSM-based B+-trees with optional LSM-based secondary indexes [2]. AsterixDB translates a computation into a directed-acyclic graph (DAG) of operators and connectors, and sends it to Hyracks for execution.

Each record in an AsterixDB dataset is identified by a unique primary key and records are hash-partitioned across the nodes on their primary keys. Each partition is locally indexed by a primary key in an LSM B+-tree, a.k.a. the primary index, and resides on its node’s local storage. AsterixDB also supports secondary indexing, including B+-tree, R-tree, and inverted indexes, partitioned in the same way as the primary index.

3 USING SIMILARITY QUERIES

In this section, we discuss similarity measures supported in AsterixDB and how users express similarity queries. We also show how users can specify indexes to expedite query processing.

3.1 Supported Similarity Measures

AsterixDB supports two similarity measures, edit distance and Jaccard, to solve string and set similarity queries. Both measures can be processed with or without indexes. Let us focus on edit distance first. It can be calculated on two strings. As an extension in AsterixDB, edit distance can also be computed between two ordered lists. For example, the edit distance between [“Better”, “than”, “I”, “expected”] and [“Better”, “than”, “expected”] is 1. This generalization is possible since a character in a text string can be viewed as an element on an ordered list if we think of a string as a collection of ordered characters.

A Jaccard value can be computed on two lists of elements. If a field type is string, a user can use a tokenization function such as “word-tokens()” to make a list of elements from the string. For example, it is possible to calculate the Jaccard similarity between two strings by tokenizing each string into a list of words. The types of the elements on a list should be the same.

If a user wishes to use their own similarity measure, they can opt to create a user-defined function (UDF). A UDF can be expressed

in AQL or SQL++ (two query languages supported by AsterixDB) or implemented as an external Java class. If the desired UDF can be expressed in AQL or SQL++, the user can create such a function using the following syntax.

```
create function similarity-cosine(x, y) {
    .....
}
```

3.2 Expressing Similarity Queries

AsterixDB provides two ways to express a similarity query in AQL or SQL++, illustrated by the example AQL in Figure 4. These queries find the record pairs from the Amazon review dataset that have similar summaries. In Figure 4(a) before the actual query, the similarity function and threshold are defined with a “set” statement. The query then uses a similarity operator “~=", which is a syntactic sugar defined for similarity functions. The “~=" operator computes the similarity between its two operands according to the “simfunction” and “simthreshold,” and returns the records that are similar. The same query can be written without using the similarity operator by a more experienced user. In Figure 4(b), the query uses the “similarity-jaccard()” function, and this query is equivalent to that in Figure 4(a). The first syntax can be easier to use since default settings for “simfunction” and “simthreshold” exist so that a user does not need to provide the two “set” statements. In addition, the user does not need to know the exact function name. Also, if the user wants to change the similarity function, they only need to change the “set” statements without changing the query itself. During query parsing and compilation, it is easy for the optimizer to detect this syntactic sugar and generate a desired optimized plan. On the other hand, the second form gives the user more direct control. There are a few variations of similarity functions in AsterixDB, e.g., one that can do early termination during the evaluation, and a user can freely choose any of them.

```
use dataverse TextStore;
set simfunction 'jaccard';
set simthreshold '0.5';
for $t1 in dataset AmazonReview
for $t2 in dataset AmazonReview
where word-tokens($t1.summary) ~= word-tokens($t2.summary)
return { 'summary1': $t1, 'summary2': $t2 }
```

(a) ~=" Notation

```
use dataverse TextStore;
for $t1 in dataset AmazonReview
for $t2 in dataset AmazonReview
where similarity-jaccard(word-tokens($t1.summary),
word-tokens($t2.summary)) >= 0.5
return { 'summary1': $t1, 'summary2': $t2 }
```

(b) Function Notation

Figure 4: AQL join on the “summary” field of the Amazon reviews using Jaccard similarity.

3.3 Using Indexes

Without an index, AsterixDB scans the whole dataset to compute the result for the given query. To expedite the execution, AsterixDB supports two kinds of inverted indexes. The first type, called “keyword index,” uses the elements of a given unordered list, and is suitable for Jaccard similarity. The two queries in Figure 4 could utilize a keyword index on the “summary” field. A keyword index can be created using the following DDL statement, where “smix” is the index name:

```
create index smix on AmazonReview(summary) type keyword;
```

The second index type is “ n -gram index,” and is suitable for edit distance. An n -gram index uses the extracted n -grams of a string as the keys, and maps those keys to their corresponding primary ids. The following is an example DDL statement to create a 2-gram index on the “reviewerName” field:

```
create index nix on AmazonReview(reviewerName) type ngram(2);
```

4 EXECUTING SIMILARITY QUERIES

In this section, we discuss how similarity queries are internally executed in AsterixDB. First, we present the execution flow for a similarity query in the presence of an index, then describe the execution flow when no index is available.

4.1 Executing Similarity Selections

We first present the execution strategy for selection queries. We use an example query to explain the execution flow for Figure 5, which computes the edit distance between a field V of a dataset and a constant C .

```
for $t1 in dataset bar
where edit-distance($t1.V, C) < 2
return {"id": $t1.id, "field": $t1.V}
```

Figure 5: A similarity-selection query.

4.1.1 Index-Based Search Execution. When running the above query on a cluster with multiple nodes, the query coordinator (*a.k.a.* cluster controller) sends a request containing the constant search key C to each participating node, since AsterixDB uses a shared-nothing architecture. Figure 6 illustrates how a similarity-selection query is executed using a secondary inverted index on a 3-node cluster. Each node contains a partitioned primary index and a local inverted index.

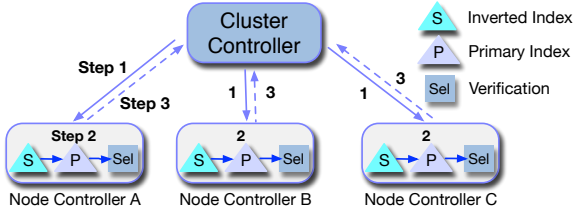


Figure 6: Parallel execution of a similarity-selection query.

If an index is available, AsterixDB runs an index-based selection plan at each node. It first gives the constant value C to the secondary inverted index. The secondary-inverted-index search generates $\langle \text{SecondaryKey}, \text{PrimaryKey} \rangle$ pairs that satisfy the T -occurrence condition, which may include false positives. It then looks up these primary keys in the primary index to fetch their corresponding records. The primary keys are sorted prior to this search to increase the chance of page cache hits in the buffer. After the primary-index search, a SELECT operator is applied to remove false positives and generate the final results. If the similarity condition is selective enough, such an index-based search plan can be much more efficient than a non-index-based plan that uses SCAN and SELECT operators. Once the local results are generated at each node, they are sent to the coordinator to be combined.

The compiler generates a non-index-based selection plan (the left part of Figure 7). The optimizer then transforms the initial plan to an index-based selection plan if there is an applicable index. We will discuss this rewriting process further in Section 5.1.

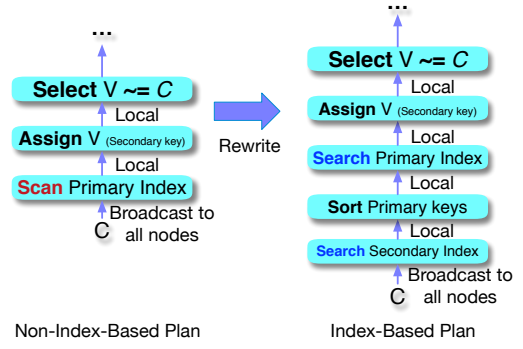


Figure 7: A similarity-selection query plan

4.1.2 Non-Index-Based Search Execution. Similar to index-based execution, when there are multiple nodes, the coordinator sends a request containing the search key C to all nodes. At each node, as there is no index on the field in the given similarity condition, AsterixDB scans the primary index, fetches all records, and verifies the similarity condition on the given field for each record. This process was depicted on the left part of Figure 7. Finally, the results will be returned to the coordinator.

4.2 Executing Similarity Joins

A join has two branches as its input. We call the first one the “outer branch” and the second one the “inner branch.” For example, in Figure 8, the AQL variable $t1$ refers to the outer branch and $t2$ refers to the inner branch.

```
for $t1 in dataset bar
for $t2 in dataset foo
where similarity-jaccard($t1.A, $t2.B) > 0.5
return {"of1": $t1.f1, "of2": $t1.f2, "A": $t1.A,
       "if1": $t2.f1, "if2": $t2.f2, "B": $t2.B}
```

Figure 8: A similarity-join query.

4.2.1 Index-Based Join Execution. Similar to the similarity-selection case, where a predicate was broadcast to all nodes, the records from the outer branch of each node are broadcast to all nodes in the similarity-join case. Figure 9 depicts how a similarity-join query is executed using a secondary inverted index on a 3-node cluster.

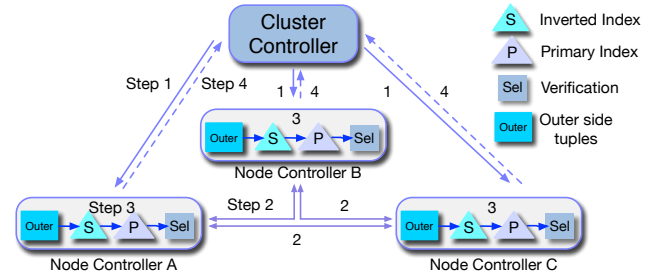


Figure 9: Parallel execution of a similarity-join query.

The coordinator sends the query request to each participating node. Each node of an outer-branch partition scans the outer-branch data and broadcasts its records to all nodes with a secondary-index partition. This broadcast replicates all records of the outer-branch on each node where the secondary-index search will be performed. Each node with an index-side partition uses the incoming outer-branch records as well as its local ones to search

its local inverted index. Once each secondary-index partition has processed all the records from the outer branch, the resulting primary keys from the search will be fed into the primary index, and a primary-index search will be executed. Again, the primary keys are sorted before the primary-index search to increase the chance of page cache hits. As before, we need to remove false positives from the index-based subplan using a SELECT operator on the original similarity condition, which is taken from the join operator. This plan is depicted on the right part in Figure 10. Finally, the results are sent to the coordinator to be combined.

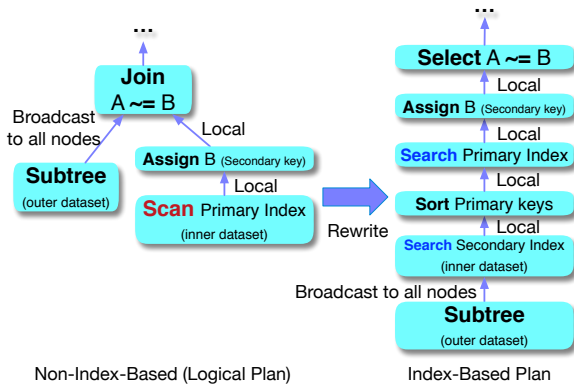


Figure 10: A similarity-join query plan.

4.2.2 Non-Index-Based Join Execution. When there is no index, a simple nested-loop join could be performed. The outer branch would feed the predicate from each record to the primary index of the inner branch. The complexity of this solution is quadratic. To avoid a costly nested-loop join, we instead adopt a three-stage algorithm [34] in AsterixDB.

Since this algorithm uses a prefix-filtering method, a global token order needs to be established to generate a prefix for each field value. This global token order can be any arbitrary order, and we choose the increasing token-frequency order, which tends to generate fewer candidate pairs [34]. The first stage computes a global token order by counting the frequency of each token in the tokenized data and sorting the tokens based on their frequencies. In the second stage, the algorithm computes a short prefix subset for each set based on the global token order produced in the first stage. Then, the record id and only the join attribute of each record are replicated and repartitioned by hashing its prefix tokens. After the repartitioning step, candidate pairs are generated by grouping the pairs by their ids, and the similarity is computed for each pair to filter out the dissimilar ones. This stage produces only similar record id pairs. Finally, the third stage rescans the inputs to fetch the rest of the record fields for these id pairs.

To apply this algorithm in AsterixDB, rather than implementing new operators and complex query plans, we chose to represent the algorithm using existing AQL constructs such as “for”, “let”, “group by”, and “order by” since this approach is more extendable in the future. In addition, if/as we improve existing operators, we do not need to modify the given AQL to utilize the improved operators. Figure 11 shows an AQL query capturing the three stages for a self-similarity join on the “summary” field of the Amazon Review dataset, using Jaccard similarity with a threshold; each step is implemented using basic AQL constructs and functions. We now discuss the details of these three stages.

Stage 1: Token Ordering is expressed in lines 11-18 of Figure 11. In this subquery we iterate over the records in the dataset. For each record, we retrieve the tokens in the “summary” field and

```

1 // -- Stage 3 --
2 for $ARevLeft in dataset('ARevs')
3 for $ARevRight in dataset('ARevs')
4 for $ridpair in
5 // -- Stage 2 --
6 for $ARevLeft in dataset('ARevs')
7 let $lenLeft := len($ARevLeft.summary)
8 let $tokensLeft :=
9   for $tokenUnranked in $ARevLeft.summary
10  for $tokenRanked at $i in
11 // -- Stage 1 --
12 for $t in dataset('ARevs')
13 let $id := $t.ARev_id
14 for $token in word-tokens($t.summary)
15 /** hash */
16 group by $tokenGrouped := $token with $id
17 order by count($id), $tokenGrouped
18 return $tokenGrouped
19 where $tokenUnranked = /** bcst */ $tokenRanked
20 order by $i
21 return $i
22 for $prefixTokenLeft in subset-collection($tokensLeft, 0,
23 prefix-len-jaccard($lenLeft, .5f) - $lenLeft + len($tokensLeft))
24
25 for $ARevRight in dataset('ARevs')
26 let $lenRight := len($ARevRight.summary)
27 let $tokensRight :=
28   for $tokenUnranked in $ARevRight.summary
29   for $tokenRanked at $i in
30 // -- Stage 1 --
31 for $t in dataset('ARevs')
32 let $id := $t.ARev_id
33 for $token in word-tokens($t.summary)
34 /** hash */
35 group by $tokenGrouped := $token with $id
36 order by count($id), $tokenGrouped
37 return $tokenGrouped
38 where $tokenUnranked = /** bcst */ $tokenRanked
39 order by $i
40 return $i
41 for $prefixTokenRight in subset-collection(
42 $tokensRight, 0, prefix-len-jaccard($lenRight, .5f))
43
44 where $prefixTokenLeft = $prefixTokenRight
45 let $sim := similarity-jaccard($tokensLeft, $tokensRight, .5f)
46 where $sim >= .5f and $ARevLeft.ARev_id < $ARevRight.ARev_id
47 group by $idLeft := $ARevLeft.ARev_id,
48 $idRight := $ARevRight.ARev_id with $sim
49 return {'idLeft': $idLeft, 'idRight': $idRight, 'sim': $sim[0]}
50
51 where $ridpair.idLeft = $ARevLeft.ARev_id and
52 $ridpair.idRight = $ARevRight.ARev_id
53 order by $ARevLeft.ARev_id, $ARevRight.ARev_id
54 return {'left': $ARevLeft, 'right': $ARevRight, 'sim': $ridpair.sim}

```

Figure 11: Three-stage set-similarity algorithm expressed in AQL for a self join on the Amazon Review (ARevs) dataset using Jaccard similarity with a threshold of 0.5.

count the number of occurrences of each token using a group-by clause. To expedite this calculation, we use a compiler hint in line 15, which suggests using hash-based aggregation instead of the default sort-based aggregation for the group-by statement, since the order of tokens at this particular step is not meaningful. Finally, we order the tokens based on their count using an order-by clause. The same subquery is repeated later, in lines 30-37, in the context of the second dataset. During the optimization, the optimizer will detect the common subquery and execute the subquery only once using a replicate operator to send the results to both outer plans. More details can be found in Section 5.4.2.

Stage 2: Record ID (RID)-Pair Generation is expressed in lines 5-50. We scan the dataset in line 6, then retrieve each token from the “summary” field. We order the tokens by the rank computed in the first stage (lines 12-23) by joining the set of tokens in one summary with the set of ranked tokens. We use a hint in line 19 that advises the compiler to use a broadcast join operator to

broadcast the ranked-tokens. Next, we order the join results by rank, stored in the variable “\$i.” We then extract the prefix tokens in line 22, and use the “prefix-len-jaccard()” built-in function to compute the length of the prefix for Jaccard similarity with a threshold of 0.5. The built-in “subset-collection()” function extracts the prefix subset of the tokens. The same process of tokenizing, ordering the tokens, and extracting the prefix tokens is done in lines 25-42 for the second stream of the dataset. We then join the two streams on their prefix tokens in line 44, and compute and verify the similarity of each joined pair. We use the built-in “similarity-jaccard()” function to compute the similarity. Since a pair of records can share more than one token in their prefixes, duplicate pairs could be produced, and they are eliminated by using a group-by clause in line 48.

Stage 3: Record Join is expressed in lines 1-4 and 51-54, which consists of two joins. The first join adds the record information for the first RID of each RID pair, while the second join adds the record information for the second.

The plan resulting from this large AQL query is shown in Figure 12. In the figure, “Hash repartition” means that a tuple is repartitioned to a corresponding node based on its hashed value. With “Hash repartition merge,” a step of merging tuples based on sort field values occurs after a “Hash repartition.” To transform a logical plan generated from a user’s similarity join query to the three-stage-similarity query plan utilizing the AQL in Figure 11, we develop a new framework called AQL+, which will be discussed in Section 5.2.

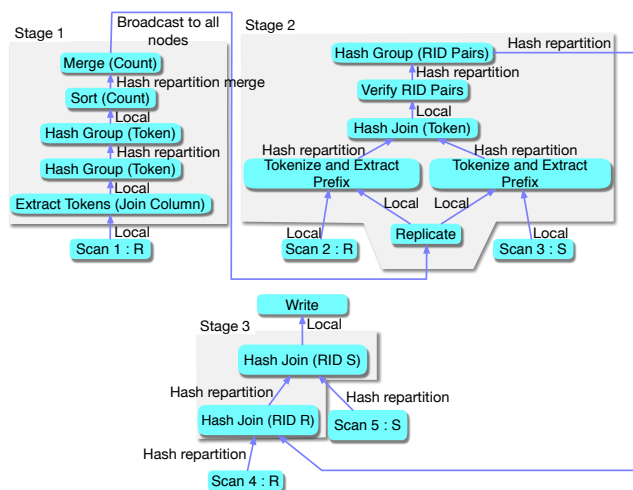


Figure 12: A logical plan of a three-stage-similarity join.

5 OPTIMIZING SIMILARITY QUERIES

In this section, we discuss how AsterixDB optimizes similarity queries and describe the AQL+ framework.

5.1 Rewriting a Similarity Query

AsterixDB uses rule-based optimization [8]. A logical plan is constructed from a given query, and each optimization rule is tried on this plan. If a rule is applicable, then the plan is transformed. A logical plan involving a dataset always starts with a primary-index scan, followed by a SELECT operator if there is one or more conditions. A non-index similarity query plan is constructed first, and an index-based transformation or a three-stage-similarity join can be introduced during the optimization.

5.1.1 Rewriting a Similarity-Selection Query. Figure 7 showed how a similarity-selection query is optimized to use an index. The left-hand side shows the original scan-based query plan, and the right-hand side shows the optimized plan. Based on a selection operator with a similarity condition (using the “~=” notation), the optimizer tries to replace the primary-index scan with a secondary-index-based search plan.

To rewrite a similarity-selection query, the optimizer first matches an operator pattern consisting of a pipeline with a SELECT operator and a PRIMARY-INDEX SCAN operator. Next, it analyzes the condition of the given SELECT operator to see if it contains a similarity condition and if one of its arguments is a constant. If so, it determines whether the non-constant argument originates from the PRIMARY-INDEX SCAN operator and whether the corresponding dataset has a secondary index on a field variable V . For each secondary index on V , it checks an index-to-function-compatibility table (Figure 13) to determine its applicability. For example, an n -gram index can be utilized for the “edit-distance()” function. The final SELECT operator filters out false positives.

Index Type	Supported Functions
n -gram	edit-distance(), contains()
keyword	similarity-jaccard()

Figure 13: Index-function compatibility table.

Corner cases: Recall that for queries using edit distance, the lower bound on the number of common q -grams (or tokens) may become zero or negative. For such a corner case, the optimizer must revert to a scan-based plan even if an index is available. For selection queries, it can foresee such cases at compile time when applying the corresponding index-rewrite rule by analyzing the constant argument in the similarity condition. When detecting a corner case, it simply stops rewriting the plan. Note that no such corner cases are possible for similarity queries based on Jaccard, because if two sets have no elements in common, then they can never reach a Jaccard similarity greater than 0. In contrast, two strings could be within a certain (large) edit distance even if the n -gram sets of the (short) strings have no common elements.

5.1.2 Rewriting a Similarity-Join Query. The basic rewriting of a similarity-join query using an index is shown in Figure 10. The optimized plan on the right-hand side uses an index-nested-loop join strategy. Similar to the rewrite for selection queries, the optimizer replaces the primary-index scan of the inner branch with a secondary-index search followed by a primary-index search. Thus, it is required that the inner branch of the join is a primary-index scan, while the outer branch could be an arbitrary operator subtree (shown as “Subtree” in the figure). In the optimized plan, the outer branch feeds into the secondary-index search operator, i.e., every record from “Subtree” will be used as a search key to the secondary index.

As in the similarity-selection case, the optimizer needs to remove false positives from the index-based subplan with a SELECT operator on the original similarity condition, which is taken from the join operator. Notice the “broadcast” connection between the outer subtree and the secondary-index search, which signifies that each partition executing the “Subtree” plan will broadcast its output stream’s records to all the secondary-index partitions. The optimizer first matches the required operator pattern consisting of a JOIN that has at least one input coming from a PRIMARY-INDEX SCAN. Next, it analyzes the join condition to make sure the similarity function has two non-constant arguments. If so, it continues

by checking if the inner argument B of the similarity condition is produced by the join input from the PRIMARY-INDEX SCAN, and whether the corresponding dataset has applicable secondary indexes. Finally, the optimizer consults the index compatibility matrix to decide whether it can rewrite the query using an index.

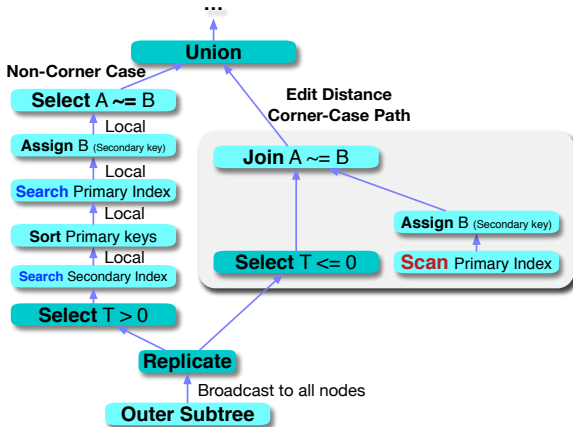


Figure 14: An optimized similarity-join query plan with the corner case.

Corner cases: For string-similarity joins using edit distance, we must modify the basic index-nested-loop join plan in Figure 10 to correctly handle corner cases. Unlike selection queries where the secondary-index search key is a constant, the secondary-index search keys for an index-nested-loop join are produced by the outer branch (“Subtree”). Join corner cases must therefore be dealt with at query runtime, as opposed to query compile time for selection queries. Figure 14 shows the modified index-nested-loop plan for correctly handling corner cases for edit distance. The main difference lies in separating the records produced by the outer subtree into two sets, one containing non-corner-case records ($T > 0$), and one containing corner-case records ($T \leq 0$). We do this by using a replicate operator above the outer subtree, followed by a selection operator on each of its two outputs to filter out the corner-case and non-corner-case records, respectively. The non-corner-case records are fed into the secondary-to-primary index plan as before, while the corner records participate in a non-index nested-loop join plan. The final query answer is the union of the results of those two joins.

5.2 AQL+ Framework

As discussed in Section 4.2.2, we need to find a way to transform a nested-loop-join plan generated from a user’s similarity-join query to a three-stage join plan. An issue is that, unlike the index-nested-loop-join optimization that adds or replaces a few operators from a nested-loop join plan, as we can see in the AQL query in Figure 11, a three-stage-similarity join query generates a large number of operators. Figure 15 shows the number of operators in a three-stage-similarity join.

Operator	Count	Operator	Count
Assign	12	Aggregate	6
Data-Scan	2	Assign	44
Join	1	Data-Scan	6
Total	15	Group	3
		Unnest	8
		Total	77

Nested-loop join plan Three-stage-similarity join plan

Figure 15: Number of operators for a nested-loop join and three-stage-similarity join plan for the same query.

Due to this complexity, it would be difficult to build an optimization rule that manually constructs these operators to transform a simple nested-loop join plan to a three-stage join plan. Instead, we develop a novel rewrite framework called “AQL+.” As shown in Figure 16, we use this framework to convert a simple logical plan generated from a user’s join query to a three-stage join plan.

Once the optimizer receives a logical plan in AQL+, it extracts the information from the logical plan and integrates it into an AQL+ query template. The generated AQL+ query can be parsed and compiled again using the AQL+ parser and translator. The result is a transformed logical plan, and the plan optimization process can then continue.

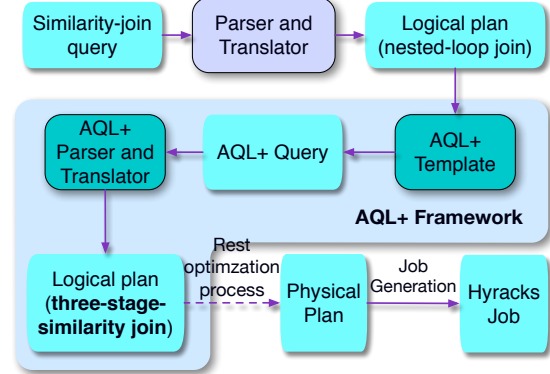


Figure 16: Execution of a similarity-join query using AQL+.

To combine the information from a logical plan and the three-stage-similarity-join AQL query template, we need to find ways to refer to the portions of the logical plan from the query template. Therefore, the AQL+ framework consists of a few AQL language extensions and the compilation of these language extensions during the optimization process. As a result, the AQL+ language is a superset of AQL. The AQL+ has three AQL extensions: Meta Variable (denoted as “\$\$”), Meta Clause (“##”), and Explicit Join (“join”). We need these extensions to refer to the logical variables and operators in the logical plan during the optimization process, since the AQL+ transformation of a given plan happens during the optimization process. This is because the optimizer only sees the logical plan and physical plan, not the original query. Since AQL itself does not have an explicit join clause, AQL+ includes one in order to express a join on two branches. For example, we use meta-variables to refer to the primary keys of the input records or variables in the similarity predicate. The usage of meta-clauses is to refer to inputs of the AQL query and to logical constructs that cannot be directly specified in AQL, such as joins. So any AQL+ template can be combined with any join input branches, where the inputs can be from any kind of subplans of other algebraic operators. In addition, to support various types of data, similarity functions, and thresholds, the similarity-join rule template uses placeholders, which are parts of the AQL+ query and are unknown until runtime. They are used for data types, similarity-specific functions, or values. For example, the “SIMILARITY” placeholder is used for built-in AQL functions, and the “THRESHOLD” placeholder is for numerical values.

Table 1: AQL+ extensions.

Extension	Symbol	Functionality
Meta Variable	\$\$	Refer to a variable in the plan
Meta Clause	##	Refer to an operator in the plan
Join Clause	join	Express an explicit join

The AsterixDB optimizer integrates the information from the given logical plan into the AQL+ query template and compiles the

resulting AQL+ query. Specifically, for a three-stage-similarity join, it needs to identify a similarity join operator that contains a Jaccard similarity join and its threshold. It also needs to get the information about the two branches of this join operator. Using the information from the join operator, the logical plan fed into this AQL+ template can be transformed to the equivalent three-stage-similarity plan. Rather than doing this transformation by introducing a number of operators by hand, we rely on the existing compilation path to generate a revised plan. This process is depicted in Figure 16; the details of this optimization will be discussed in the next subsection.

For the similarity join query in Figure 11, the optimizer will generate an equivalent AQL+ template and use it to transform a simple query during the rule-rewrite phase. In this way, the simple query of Figure 4(a) can be transformed to the query in Figure 11 during the optimization process. Figure 17 shows a part of the AQL+ template that generates a three-stage-similarity-join plan. Here, we can see that actual dataset-scans are replaced with meta-clauses and a meta-variable ($\$LEFTPK_3$) is used to refer to the primary key of an incoming record in the given logical plan. Join-clauses are used to join two meta-clauses.

```

1  //---Stage3---
2  join( (##RIGHT_1),
3        ( join( (##LEFT_1) ,
4              // --- Stage 2 ---
5                ( join( (##LEFT_2
6                  .....
7                    //--- Stage1---
8                    ##LEFT_3
9                    let $sid := $LEFTPK_3
10                   for $token in TOKENIZER($RIGHT_3)
11                   /*+ hash */
12                   group by $tokenGrouped := $token with $sid
13                   .....

```

Figure 17: A part of three-stage-similarity-join algorithm expressed in AQL+.

In addition, the AQL+ framework can be applied to transform multi-way-similarity join plans as well because of its power to handle a logical plan iteratively. Similar to non-similarity-join cases, multi-way-similarity joins can be transformed sequentially without a limitation. For instance, Figure 18 shows a similarity-join plan involving four datasets. The join between first two datasets, R and S , has already been transformed into a three-stage-similarity join plan. This branch will act as the outer branch when the optimizer processes the next join operator on the third dataset T .

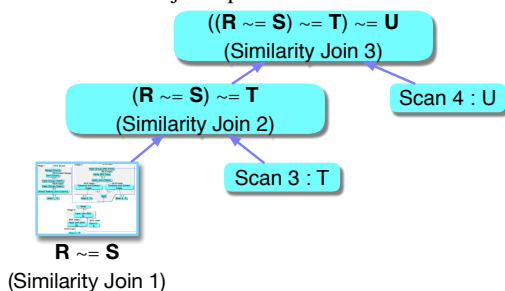


Figure 18: Rewriting a multi-way-similarity-join plan on four datasets.

It should be noted that AQL+ is a general extension framework, not only for similarity queries, and it can be used to support a transformation using AQL during the compilation process.

5.3 Optimization Rule For Similarity Queries

As discussed before, optimization in AsterixDB is rule-based [8]. Once the Algebricks layer receives a compiled plan from an AQL

query, it first optimizes the given plan logically. Then Algebricks sets up physical operators for each logical operator. After that, the physical optimization phase begins. When it is finished, a Hyracks job is created and executed. During the logical and physical optimization, there are a number of rule sets that are applied sequentially. A rule can be assigned to multiple rule sets. Based on the configuration of a rule set, each rule can be applied repeatedly until no rule in the set can transform the plan.

To apply the similarity-query optimization framework to this optimization path, we create a new rule set for the AQL+ framework and similarity queries. The rule set includes a *similarity join rule* (SJR) along with a handful of other rules that need to be applied after SJR is applied. As described earlier, the main functionality of AQL+ is a transformation using a complex AQL template to re-generate a logical plan while maintaining the current surrounding plan as part of the new plan. SJR first analyzes the conditions of each join operator. If its condition includes a similarity predicate, it applies the AQL+ template to the plan to generate an AQL+ query. Then it compiles the query into a new logical Algebricks plan. Some parts of the plan were already optimized if they belonged to the original incoming plan. However, most part of the plan is not optimized yet, since the three-phase plan was just compiled and has not gone through the optimization process before the SJR rule set. Therefore, the newly generated plan needs to go through some of the earlier optimization rules again. This re-application process is not necessary for non-similarity queries, since the plan generated from non-similarity queries is not transformed in the SJR rule set. Therefore, we need to ensure that the similarity-join rule set is only applied to similarity-join queries. The benefit of this approach is that the optimization for similarity queries can be processed without interfering with non-similarity queries. This approach also gives a chance to the newly generated similarity-query plan to reach the same level of transformation once the similarity rule set has finished its work.

5.4 Improvements

We discuss two improvements to similarity query processing, which can be applied to non-similarity query processing as well.

5.4.1 Surrogate Index-Nested-Loop-Join. A drawback of an index-nested-loop join using a local secondary index is the need to broadcast the outer side data to all secondary-index partitions as explained in Section 4. For example, during an execution of the AQL query in Figure 8, the outer side needs to broadcast join key field “A,” as well as “ $f1$ ” and “ $f2$ ” field. If there are more fields in the return clause, the broadcasting cost will be increased as well. This broadcast step is a direct consequence of the co-partitioning of each secondary index with its primary index. Also a secondary-inverted-index search can generate multiple pairs of results for the same primary key, as there can be multiple entries of the secondary keys for the same primary key; thus, we also want to reduce the sorting cost between the secondary-index search and primary index search. We can reduce the cost by only sending the secondary-key fields together with a compact surrogate for each outer-side record, so that we can later use the surrogates to obtain the surviving original records. This idea is reminiscent of semi-join optimization in distributed databases [40].

Figure 19 shows a surrogate-based index-nested-loop-similarity join plan. Notice the PROJECT operator that follows the REPLICATE operator after the outer subtree, which eliminates all non-essential fields from the outer side. The optimizer filters out the “ $f1$ ” and “ $f2$ ” fields since the search key is “A.” In addition, since

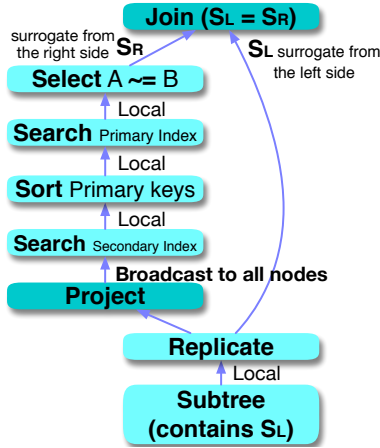


Figure 19: Surrogate index-nested-loop-join plan.

the same subtree is used twice in the plan, a REPLICATE operator is introduced to reduce the subtree calculation time. We will discuss this optimization in-depth in the next subsection. After the secondary-to-primary index search, we must use the surrogates from the outer side to obtain their complete records. As shown in the figure, we resolve the surrogates via a top-level join of the original outer subtree with the indexed nested-loop subtree (after removing false positive matches). Since the top-level join is an equi-join on the surrogates S_L and S_R , it can be executed efficiently in parallel, e.g., using a hash join.

5.4.2 Materializing/Reusing Shared Subplans. As shown in a simplified version of a three-stage-similarity join in Figure 20, in case of the three-stage-similarity self join, the dataset R may need to be scanned three to four times. For this case, we could simply execute the original data-scan operation four times. However, if the two branches of this join result from a complex computation from a subquery, it would be expensive to compute the result of the subquery many times. To minimize the cost, AsterixDB materializes the common subplan and reuses it several times.

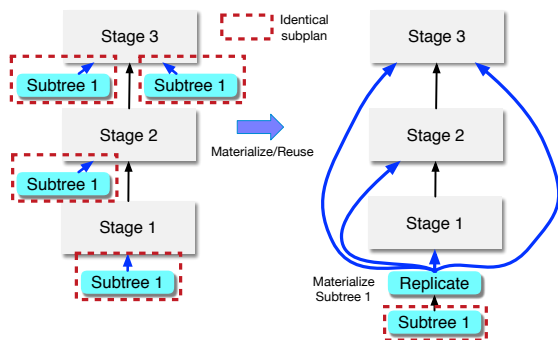


Figure 20: Materializing and reusing a subtree of a three-stage-similarity self join.

6 EXPERIMENTS

We have conducted an experimental evaluation of our approach in AsterixDB using large, real data sets. We used an 8-node cluster to host an AsterixDB (0.9.2) instance, where each node ran Ubuntu with a Quadcore AMD Opteron CPU 2212 HE (2.0GHz), 8GB RAM, 1 GB Ethernet NIC, and two 7,200 RPM SATA hard drives. Each dataset was horizontally partitioned into 16 partitions (2

per node) based on primary keys to provide full I/O parallelism. Table 2 shows the AsterixDB configuration parameters.

Table 2: AsterixDB parameters for the experiments.

Parameter	Value
Global memory budget per node	6GB
Budget for in-memory components per dataset	1.5GB
Data page size	128KB
Disk buffer cache size	2GB
Sort buffer size	128MB
Join buffer size	128MB
Group-by buffer size	128MB

6.1 Datasets

Different similarity functions were used for different types of data. Edit distance is more suitable for short string fields, while Jaccard is more suitable for long fields with many elements. To evaluate AsterixDB for different similarity functions, we used three datasets with different characteristics, as shown in Table 3. The Amazon Review dataset, discussed in earlier sections, included Amazon product reviews [23]. The Reddit Submission dataset included Reddit postings for about eight years. The Twitter dataset had 1% of US tweets for three months obtained via Twitter’s public API. When imported into AsterixDB, each data set had an additional auto-generated primary key field, as AsterixDB requires that each dataset must have a primary key. Other than this field, we did not define more fields in the schema. This gave us a lot of flexibility to import any datasets into AsterixDB. The dataset size in AsterixDB was greater than the raw data size, since each record included the information about each field. For example, for a string field, its type needs to be included in addition to its value.

Table 3: Dataset properties.

Dataset	AmazonReview	Reddit	Twitter
Content	Amazon product reviews	Reddit postings	Tweets
Number of Records	83.68M	196M	155M
Data Period	1996 - 2014	01/2006 - 08/2015	06/2016 - 08/2016
Raw Data Format	JSON	JSON	JSON
Raw Data Size	55 GB	252 GB	465 GB
Dataset Size	60.6 GB	320 GB	582 GB
Fields used	summary, reviewerName	title, author	text, user.name

Table 4 shows the characteristics of the fields of the datasets. The minimum character length and minimum word count of the fields were 0. The first three fields were used for edit distance, while the last three fields were used for Jaccard.

Table 4: The characteristics of the fields.

Field	Avg char count	Max char count	Avg word count	Max word count
AmazonReview.reviewerName	10.3	49	1.7	14
Reddit.author	24.3	275	4.1	32
Twitter.user.name	10.6	20	1.7	10
AmazonReview.summary	22.8	361	4.0	44
Reddit.title	1,056.2	400K	1,173	20K
Twitter.text	62.5	140	9.7	70

6.2 Index Size

We built a keyword index for Jaccard similarity queries and a 2-gram index for edit distance queries. To measure the execution time of basic exact match queries on the same fields as a baseline, we also built a B+ tree index on the search fields. Table 5 shows

the index sizes for the Amazon Review dataset and the time it took to create each index. An n -gram index took much more space than a B+ tree or keyword index as it had more secondary keys per record. For instance, a 2-gram index on the “reviewerName” field took 15.6GB of disk space, which was about 25% of the original dataset size. The size of a keyword index was also greater than a B+ tree index on the same field since there are many secondary keys per record. For a given type of index, the construction time was roughly proportional to the size of an index. In each case, the dataset itself was also stored in a primary B+ tree index.

Table 5: Index size and build time for Amazon reviews.

Field	Index Type	Size (GB)	Build Time (s)
Dataset itself	B+ tree	60.6	1,563
reviewerName	B+ tree	2.7	223
reviewerName	2-gram	15.6	1,441
summary	B+ tree	3.5	275
summary	keyword	5.4	573

6.3 Selection Queries

To measure the performance of similarity-selection queries, we first created a search value set that contained 10,000 random unique values extracted from the search field. For Jaccard queries, we ensured that the minimum number of words in each value in the set was 3. For edit distance queries, the minimum length of characters in each value was 3. For each similarity threshold, we randomly chose a search value from the set for each query, sent 100 such queries to the cluster, and measured the average execution time. The performance baseline for comparison purposes was an equality-condition query that used the same value for the given field. The query template in Figure 21 below was used to measure the average execution time. “Simfunction” and “simthreshold” in the queries were replaced with a specific similarity function and a threshold. “V” was the given field and “C” was the random value from the above set.

```
count ( for $o in dataset X
  where @simfunction($o.V, C) >= @simthreshold
  return {"oid":$o.id, "v":$o.V} );
```

Figure 21: Similarity-selection query template.

6.3.1 Jaccard Similarity. For each of the three datasets we ran similarity queries using Jaccard similarity on suitable fields using different thresholds: 0.2, 0.5, and 0.8. Figure 22(a) shows the results. We see that the average execution time for similarity selection queries decreased as the threshold increased in case of index-based plans. For example, it took the index-based method 67.6 seconds to conduct a Jaccard query with a threshold of 0.2, while it only took 25.5 seconds to execute a query with a threshold of 0.5. If there was no applicable index, both similarity and exact-match queries showed a high execution time as each record had to be read from the primary index and that scan time was a dominant factor in the overall execution time. We can also see the overhead of the similarity query versus the exact-match query for all the thresholds since it takes more time to calculate a Jaccard value than to get the result of an exact match. This overhead decreased as the threshold increased; this is because we applied certain optimizations such as early termination and pruning based on string lengths, which significantly reduced the cost of computing the similarity.

When the threshold was low, the times were similar for both index-based and non-index-based queries. This is because the candidate set size using T -occurrence for index-based queries was quite large when the threshold was low. This can be seen in

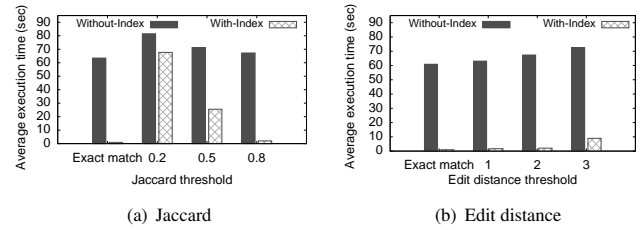


Figure 22: Execution time of selection queries on Amazon reviews.

Table 6. As the number of candidates increased, the search time increased due to the need for a primary-index lookup for each candidate.

Table 6: Candidate size and the final result size for the indexed-select query for Amazon reviews in Figure 22(a).

Jaccard Threshold	Actual Record Count (B)	Candidate Set Record Count (C)	Ratio (B/C)
0.2	559,167	8,298,473	6.7%
0.5	12,260	660,016	1.9%
0.8	36	12,420	0.3%

6.3.2 Edit Distance. We measured the average execution time of an edit distance selection query using different thresholds, namely 1, 2, and 3. Figure 22(b) shows the results. As the threshold increased, the execution time increased. The reason is similar to the case of Jaccard queries: the candidate set size using T -occurrence increased as the threshold increased. It took the index-based method 2 seconds to run a selection query with a threshold of 2; it took 8.9 seconds to run a query with a threshold of 3. We can also see that the execution time of non-index-based edit distance queries increased as the threshold increased for the same reason as described above.

6.4 Join Queries

To measure the performance of similarity join queries, we ran self-join queries on the three datasets. Specifically, the query template in Figure 23 was used to measure the average execution time as in the similarity-selection query case. Here, V is the field on which we applied a similarity function and id is the primary key field.

```
count ( for $o in dataset X
  for $i in dataset X
  where @simfunction($o.V, $i.V) >= @simthreshold
  and $o.f1 = C and $o.id < $i.id
  return {"oid":$o.id} );
```

Figure 23: Similarity-join query template.

6.4.1 Varying Threshold. We first extracted certain number of records from the outer branch of the join to limit its input. For each query, we chose 10 random records from the outer branch. In the query template in Figure 23, a field named $f1$ was used to specify such a limit. For Jaccard join queries, we used different thresholds, namely 0.2, 0.5, and 0.8. For edit distance queries, we used thresholds of 1, 2, and 3. When there was no applicable index, AsterixDB chose to employ the three-stage-similarity-join plan for Jaccard queries. The results are shown in Figures 24(a) and 24(b). The trends were similar to those of selection queries except for the exact-match join, which significantly outperformed both Jaccard

and edit distance joins since it used a hash join, where the join keys were broadcast to multiple nodes.

Regarding the compilation overhead of AQL+, we observed that the average overhead of generating a new logical three-stage-similarity-join plan using AQL+ for the queries in Figure 24(a) was around 50 ms, and it took around 500 ms to optimize that plan. The overall compilation time of the three-stage-similarity-join query was around 900 ms.

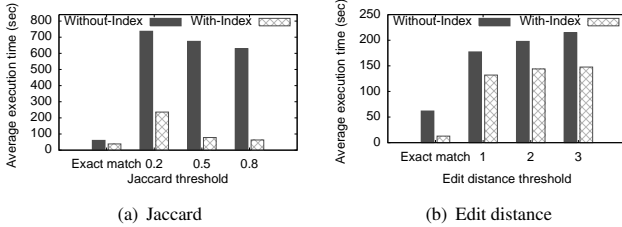


Figure 24: Execution time of join queries on Amazon reviews.

6.4.2 Varying Record Number. For a Jaccard join query, its execution time was smallest when the threshold was 0.8. In this experiment, we varied the number of records from the output branch and fixed the threshold at 0.8. The times for the non-index-nested-loop self-join, index-nested-loop self-join, and three-stage-similarity self-join on the Amazon Review dataset are shown in Figure 25(a). We increased the number of output records from the outer branch and measured the resulting execution time of each join. First, we see that the execution time of non-index-nested-loop self-join was already highest for 200 records and increased drastically compared to other two types of joins. Once the number of output records from the outer branch reached around 400, the three-stage-similarity join began to outperform the index-nested-loop join. This is because the time for the index-nested-loop join is proportional to the number of records fed to the secondary-index search, as it needs to deal with each record at a time. In contrast, for the three-stage-similarity join, most of the time is spent on global-token-order generation in the first stage. Once this is generated and broadcast to all the nodes, hash joins in stage 2 and 3 can deal with the incoming records efficiently, since each join key is sent to only one node. This benefit is visible in the figure. For instance, the time for the three-stage-similarity join for 800 records was 619 seconds, while it was 674 seconds for 1,000 records. This result shows only 55 seconds of increase, while the execution-time difference for the index-nested-loop joins going from 800 to 1,000 records was 384 seconds.

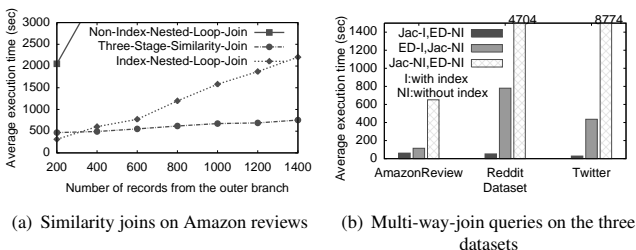


Figure 25: Execution time of join queries.

6.4.3 Multi-Way Join Queries. So far, we have used only one similarity condition per similarity query. Next, we used two similarity conditions in a query and varied the order of the two conditions. The query template in Figure 26 was used to measure

the average execution time. The dataset Y and the field $f1$ were used to limit the number of records from the outer branch.

```

count ( for $p in dataset Y
        for $o in dataset X
        for $i in dataset X
        where $p.f1 = $o.f1 and $p.id = C
        and @simfunction1($o.V1, $i.V1) >= @simthreshold1
        and @simfunction2($o.V2, $i.V2) <= @simthreshold2
        and $o.id < $i.id
        return {"oid":$o.id, "iid":$i.id} );

```

Figure 26: Multi-way-join query template.

Each query had an equi-join and a similarity join with two conditions, including a Jaccard condition with a threshold of 0.8 and an edit distance condition with a threshold of 1. For the equi-join, we used an index-nested-loop join to fetch output records quickly. Also, this join was used to limit the number of output records from the outer branch fed into the similarity join. Then, Jaccard similarity and edit distance conditions were applied. If we applied the Jaccard condition first, the Jaccard join will be followed by the edit distance condition in a SELECT operator. For both similarity conditions, we used an index-based method for the first condition and a non-index-based method for the second. Figure 25(b) shows that the performance was the best when the index-based-Jaccard join was conducted first, as there were no corner cases for Jaccard similarity. This order generated fewer candidates than applying the index-based edit distance predicate first. In contrast, for the edit distance case, it needed to augment the corner-case path in the logical plan, thus generated more candidates. In addition, it should be noted that other queries showed similar patterns for all the three datasets as well. That is, the average execution time of a similarity query was proportional to the size of datasets when the result cardinality was similar.

6.5 Scalability Tests

6.5.1 Scale-Out. For the scale-out experiment, we used four clusters with different sizes, namely 1, 2, 4, and 8 nodes. When we doubled the number of nodes in a cluster, we also doubled the data size to store the same amount of data per node. Thus, the 1-node cluster had 12.5% of our original data set size, the 2-node cluster had 25%, and the 4-node cluster had 50% of the data. The 8-node cluster contained the original dataset, where the data size was 100%. In other words, each node had 12.5% of the original dataset. Ideally, the response-time graph would show a flat line per query. As the number of nodes increased, the queries were handled as expected, as shown in Figure 27(a). We can see some variance in the case of the Jaccard-similarity join without an index; in the three-stage-similarity join, the global token order generated in stage 1 of the join needed to be broadcast to all the nodes. Therefore, as the number of nodes increased, the communication cost increased as well. This gap was the greatest between 1 node and 2 nodes, since that was where we first incurred the communication cost of global-token-order propagation. However, the execution time increase was not high between 2 nodes and 4 nodes. Between 4 nodes and 8 nodes, we can see the trend as well. Once the communication cost was accounted for, the execution time of three-stage-similarity joins was quite scalable.

6.5.2 Speed-Up. For the speed-up experiment, we also used four cluster sizes (1, 2, 4, and 8 nodes) with each cluster size

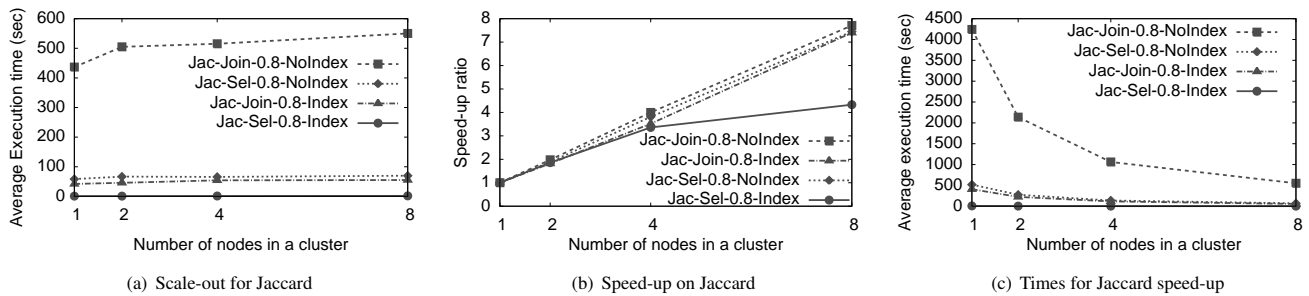


Figure 27: Scale-out and speed-up queries on Amazon reviews.

being given the entire (100%) data set. Figure 27(b) shows that the speed-up was proportional to the number of nodes. The speed-up for the index-based-Jaccard-selection query with a threshold 0.8 was less than that of the other Jaccard queries. This was because its execution time was already less than a few seconds on the 1-node cluster, and there was a basic overhead for each cluster such as communication cost. In particular, the execution time of that query on the 1-node cluster was 6.5 seconds, and its execution time on the 8-node cluster was 1.5 seconds. Figure 27(c) shows the execution time of the same queries on each cluster.

7 CONCLUSIONS

In this paper, we presented the support for similarity queries in Apache AsterixDB, a parallel data management system. We described the entire life cycle of a similarity query in the system, including the query language, indexing, execution plans with or without index, and plan rewriting to optimize the execution. Our solution leverages the existing infrastructure of AsterixDB, including its operators, query engine, and rule-based optimizer. We presented an experimental study based on several large, real data sets on a parallel computing cluster to evaluate the proposed techniques, and showed their efficacy and performance to support similarity queries on large data sets using parallel computing.

Acknowledgments The Apache AsterixDB project has been supported by an initial UC Discovery grant, by NSF IIS awards 0910989, 0910859, 0910820, 0844574, and by NSF CNS awards 1305430 and 1059436. This work was also sponsored by the National Science Foundation of China under grants 61572373, 60903035, and 61472290, and by the National High Technology Research and Development Program of China under grant 2017YFC08038. The project has received industrial support from Amazon, eBay, Facebook, Google, HTC, Infosys, Microsoft, Oracle Labs, and Yahoo! Research.

REFERENCES

- [1] Peter Christen. 2012. *Data matching: concepts and techniques for record linkage, entity resolution, and duplicate detection*. SSBM.
- [2] Alsubaiee et al. 2014. Storage Management in AsterixDB. In *Proceedings of the 2014 VLDB Conference*.
- [3] Bayardo et al. 2007. Scaling up all pairs similarity search. In *Proceedings of the 2007 WWW Conference*.
- [4] Borgatti et al. 2009. Network analysis in the social sciences. *Science* (2009).
- [5] Behm et al. 2009. Space-Constrained Gram-Based Indexing for Efficient Approximate String Search. In *Proceedings of the 2009 ICDE Conference*.
- [6] Borkar et al. 2011. Hyracks: A flexible and extensible foundation for data-intensive computing. In *Proceedings of the 2011 ICDE Conference*.
- [7] Bouros et al. 2012. Spatio-textual similarity joins. *Proceedings of the VLDB Endowment* (2012).
- [8] Borkar et al. 2015. Algebricks: A Data Model-Agnostic Compiler Backend for Big Data Languages. In *Proc. of the ACM Symp. on Cloud Computing*.
- [9] Ciaccia et al. 1997. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In *Proceedings of the 1997 VLDB Conference*.
- [10] Chaudhuri et al. 2006. Data Debugger: An Operator-Centric Approach for Data Quality Solutions. *IEEE Data Eng. Bull.* (2006).
- [11] Deng et al. 2014. Massjoin: A mapreduce-based method for scalable string similarity joins. In *Proceedings of the 2014 ICDE Conference*.
- [12] Deng et al. 2014. A pivotal prefix based filtering algorithm for string similarity search. In *Proceedings of the 2014 SIGMOD Conference*.
- [13] Doukeridis et al. 2014. A survey of large-scale analytical query processing in MapReduce. *Proceedings of the VLDB Endowment* (2014).
- [14] Feng et al. 2012. Trie-join: a trie-based method for efficient string similarity joins. *Proceedings of the VLDB Endowment* (2012).
- [15] Gravano et al. 2001. Approximate String Joins in a Database (Almost) for Free. In *Proceedings of the 2001 VLDB Conference*. 491–500.
- [16] Gravano et al. 2003. Text joins in an RDBMS for web data integration. In *Proceedings of the 2003 WWW Conference*.
- [17] Jokinen et al. 1991. Two algorithms for approximate string matching in static texts. In *International Symposium on Mathematical Foundations of Computer Science*.
- [18] Jiang et al. 2014. String similarity joins: An experimental evaluation. *Proceedings of the VLDB Endowment* (2014).
- [19] Li et al. 2007. VGRAM: Improving Performance of Approximate Queries on String Collections Using Variable-Length Grams. In *Proceedings of the 2012 VLDB Conference*. <http://www.vldb.org/conf/2007/papers/research/p303-li.pdf>
- [20] Li et al. 2008. Efficient Merging and Filtering Algorithms for Approximate String Searches. In *Proceedings of the 2008 ICDE Conference*.
- [21] Metwally et al. 2012. V-smart-join: A scalable mapreduce framework for all-pair similarity joins of multisets and vectors. *Proceedings of the VLDB Endowment* (2012).
- [22] Mann et al. 2014. PEL: Position-Enhanced Length Filter for Set Similarity Joins. In *Grundlagen von Datenbanken*.
- [23] McAuley et al. 2015. Inferring networks of substitutable and complementary products. In *Proceedings of the 2015 SIGKDD Conference*.
- [24] Mann et al. 2016. An empirical evaluation of set similarity join techniques. *Proceedings of the VLDB Endowment* (2016).
- [25] Minghe et al. 2016. String similarity search and join: a survey. *Frontiers of Computer Science* (2016).
- [26] Qin et al. 2011. Efficient exact edit similarity query processing with the asymmetric signature scheme. In *Proceeding of the 2011 SIGMOD Conference*.
- [27] Rahm et al. 2000. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.* (2000).
- [28] Ribeiro et al. 2011. Generalizing prefix filtering to improve set similarity joins. *Information Systems* (2011).
- [29] Sarawagi et al. 2004. Efficient set joins on similarity predicates. In *Proceedings of the 2004 SIGMOD Conference*.
- [30] Silva et al. 2010. The similarity join database operator. In *Proceedings of the 2010 ICDE Conference*.
- [31] Silva et al. 2012. Exploiting mapreduce-based similarity joins. In *Proceedings of the 2012 SIGMOD Conference*.
- [32] Silva et al. 2015. Similarity Joins: Their implementation and interactions with other database operators. *Information Systems* (2015).
- [33] Sun et al. 2017. Dima: A Distributed In-Memory Similarity-Based Query Processing System. *Proceedings of the VLDB Endowment* (2017).
- [34] Vernica et al. 2010. Efficient Parallel Set-Similarity Joins Using MapReduce. In *Proceedings of the 2010 SIGMOD Conference*.
- [35] Wang et al. 2012. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In *Proceedings of the 2012 SIGMOD Conference*.
- [36] Wang et al. 2013. Scalable all-pairs similarity search in metric spaces. In *Proceedings of the 2013 ACM SIGKDD Conference*.
- [37] Wang et al. 2013. VChunkJoin: An efficient algorithm for edit similarity joins. *KDE, IEEE Transactions on* (2013).
- [38] Xiao et al. 2008. Ed-Join: An Efficient Algorithm for Similarity Joins With Edit Distance Constraints. In *Proceedings of the 2008 VLDB Conference*.
- [39] Xiao et al. 2008. Efficient similarity joins for near duplicate detection. In *Proceedings of the 2008 WWW Conference*.
- [40] Donald Kossmann. 2000. The state of the art in distributed query processing. *ACM Comput. Surv.* (2000).

L-Store: A Real-time OLTP and OLAP System*

Mohammad Sadoghi[†], Souvik Bhattacharjee[‡], Bishwaranjan Bhattacharjee[#], Mustafa Canim[#]

[†]Exploratory Systems Lab

[†]University of California, Davis

[‡]University of Maryland, College Park

[#]IBM T.J. Watson Research Center

ABSTRACT

To derive real-time actionable insights from the data, it is important to bridge the gap between managing the data that is being updated at a high velocity (i.e., OLTP) and analyzing a large volume of data (i.e., OLAP). However, there has been a divide where specialized solutions were often deployed to support either OLTP or OLAP workloads but not both; thus, limiting the analysis to stale and possibly irrelevant data. In this paper, we present Lineage-based Data Store (L-Store) that combines the real-time processing of transactional and analytical workloads within a single unified engine by introducing a novel update-friendly lineage-based storage architecture. By exploiting the lineage, we develop a contention-free and lazy staging of columnar data from a write-optimized form (suitable for OLTP) into a read-optimized form (suitable for OLAP) in a transactionally consistent approach that supports querying and retaining the current and historic data.

1 INTRODUCTION

We are witnessing an architectural shift and divide in database community. The first school of thought emerged from an academic conjecture that “one size does not fit all” [37] (i.e., *advocating specialized solutions*), which has led to manifolds of innovations over the last decade in creating specialized and subspecialized database engines geared toward various niche workloads and application scenarios [5, 9, 12, 22, 28, 29, 37, 38]. This school has motivated major commercial database vendors such as Microsoft to focus on building novel specialized engines offered as loosely integrated engines, namely, Hekaton in-memory engine [9] and Apollo column store engine [19], within a single umbrella of database portfolio. Notably, recent efforts are focused on a tighter real-time integration of Hekaton and Apollo engines [17]. It has inspired Oracle to push the boundary of the basic premise that “one size does not fit all” as far as data representation is concerned and has led Oracle to develop a dual-format technique [15] that maintains two tightly integrated representation of data (i.e., two copies of the data) in a transactionally consistent manner.

However, the second school of thought, supported by both academia (e.g., [2, 6, 7, 13, 16, 24]) and industry (e.g., SAP [10], IBM DB2 BLU [29], and IBM Wildfire [4]) have revisited the aforementioned fundamental premise and advocates a generalized solution. Proponents of this idea, rightly in our view, make the following arguments. First, there is a tremendous cost in building and maintaining multiple engines from both the perspective of database vendors and users of the systems (e.g., application development and deployment costs). Second, there is a compelling case to support real-time decision making on the latest version of the data [27] (likewise supported by [15, 17]), which may not be feasible across loosely integrated engines that are connected through

the extract-transform load (ETL) process. Closing this gap may be possible, but its elimination may not be feasible without solving the original problem of unifying OLTP and OLAP capabilities or without being forced to rely on ad-hoc approaches to bridge the gap in hindsight. We argue that the separation of OLTP and OLAP capabilities defers solving the actual challenge of real-time analytics. Third, combining real-time OLTP and OLAP functionalities remains as an important basic research question, which demands deeper investigation even if it is purely from the theoretical standpoint.

In this dilemma, we support the latter school of thought (i.e., *advocating a generalized solution*) with the goal of undertaking an important step to study the entire landscape of single engine architectures and to support both transactional and analytical workloads holistically (i.e., “one size fits all”). In this paper, we present Lineage-based Data Store (L-Store) with a novel update-friendly lineage-based storage architecture to address the conflicts between row- and column-major representation. This is achieved by developing a contention-free and lazy staging of columnar data from write optimized into read optimized form in a transactionally consistent manner without the need to replicate data, to maintain multiple representation of data, or to develop multiple loosely integrated engines that sacrifices real-time capabilities.

To further disambiguate our notion of “one size fits all”, in this paper, we restrict our focus to real-time relational OLTP and OLAP capabilities. We define a set of architectural characteristics for distinguishing the differences between existing techniques. First, there could be a single product consisting of multiple loosely integrated engines that can be deployed and configured to support either OLTP or OLAP. Second, there could be a single engine as opposed to having multiple specialized engines packaged in a single product. Third, even if we have a single engine, then we could have multiple instances running over a single engine, where one instance is dedicated and configured for OLTP workloads while another instance is optimized for OLAP workloads, in which these instances are assumed to be connected using an ETL process. Finally, even when using the same engine running a single instance, there could be multiple copies or representations (e.g., row vs. columnar layout) of the data, where one copy (or representation) of the data is read optimized while the second copy (or representation) is write optimized.

In short, we develop L-Store, an important first step towards supporting real-time OLTP and OLAP processing that faithfully satisfies our definition of *generalized solution*, and, in particular, we make the following contributions:

- Introducing an update-friendly lineage-based storage architecture that enables a contention-free update mechanism over a native multi-version, columnar storage model in order to lazily and independently stage stable data from a write-optimized columnar layout (i.e., OLTP) into a read-optimized columnar layout (i.e., OLAP)
- Achieving (at most) 2-hop away access to the latest version of any record (preventing read performance deterioration for point queries)

*Work by S. Bhattacharjee was performed as part of a summer internship at IBM T.J. Watson Research Center under M. Sadoghi’s mentorship.

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

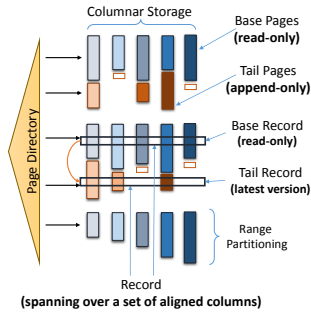


Figure 1: Overview of the lineage-based storage architecture.

- Contention-free merging of only stable data, namely, merging of the read-only base data with recently committed updates (both in columnar representation) without the need to block ongoing or new transactions by relying on the lineage
- Contention-free page de-allocation (upon the completion of the merge process) using an epoch-based approach without the need to drain the ongoing transactions
- A first of its kind comprehensive evaluation to study the leading architectural storage design for concurrently supporting short update transactions and analytical queries (e.g., an in-place update with a history table architecture and the commonly employed main and delta stores architecture)

2 UNIFIED ARCHITECTURE

The divide in the big data community is partly attributed to the storage conflict pertaining to the representation of transactional and analytical data. In particular, transactional data requires write-optimized storage, namely the row-based layout, in which all columns are co-located (and preferably uncompressed for in-place updates). This layout improves point update mechanisms, since accessing all columns of a record can be achieved by a single I/O (or few cache misses for memory-resident data). In contrast, to optimize the analytical workloads (i.e., reading many records), it is important to have read-optimized storage, i.e., columnar layout in highly compressed form. The intuition behind having columnar layout is due to the observation that most analytical queries tend to access only a small subset of all columns [1]. Thus, by storing data column-wise, we can avoid reading irrelevant columns (i.e., reducing the raw amount of data read) and avoid polluting processor’s cache with irrelevant data, which substantially improve both disk and memory bandwidth, respectively. Furthermore, storing data in columnar form improves the data homogeneity within each page, which results in an overall better compression ratio.

2.1 L-Store Storage Overview

To address the dilemma between write- and read-optimized layouts, we develop L-Store. As demonstrated in Figure 1, the high-level architecture of L-Store is based on a native multi-version, columnar layout (i.e., data across columns are aligned to allow implicit re-construction), where records are (virtually) partitioned into disjoint ranges (also referred to as update range). Records within each range span a set of read-only, compressed pages, which we refer to them as the *base pages*. More importantly, for every range of records, and for each updated column within the range, we maintain a set of append-only pages to store the latest updates, which we refer to them as the *tail pages*. Anytime a record is updated in base pages, a new record is appended to its corresponding tail pages, where there are explicit values only for the updated columns (non-updated columns are preassigned a special null value when a page is first allocated). We refer to the records in base pages as the *base records* and the records in tail pages as the *tail records*. Each record (whether falls in base

or tail pages) spans over a set of aligned columns (i.e., no join is necessary to pull together all columns of the same record).¹

A unique feature of our lineage-based architecture is that tail pages are strictly append-only and follow a write-once policy. In other words, once a value is written to tail pages, it will not be over-written even if the writing transaction aborts. The append-only design together with retaining all versions of the record substantially simplifies low-level synchronization and recovery protocol and enables efficient realization of multi-version concurrency control. Another important property of our lineage-based storage is that all data are represented in a common unified form; there are no ad-hoc corner cases. Records in both base and tail pages are assigned record-identifiers (RIDs) from the same key space. Therefore, both base and tail pages are referenced through the database page directory using RIDs and persisted identically. Therefore, at the lower-level of the database stack, there is absolutely no difference between base vs. tail pages or base vs. tail records; they are presented and maintained identically.

To speed up query processing, there is also an explicit linkage (forward and backward pointers) among records. From a base record, there is a forward pointer to the latest version of the record in tail pages. The different versions of the same records in tail pages are chained together to enable fast access to an earlier version of the record. The linkage is established by introducing a table-embedded indirection column that stores forward pointers for base records and backward pointers for tail records (i.e., RIDs).

The final aspect of our lineage-based architecture is a periodic, contention-free merging of a set of base pages with its corresponding tail pages. This is performed to consolidate base pages with the recent updates and to bring base pages forward in time (i.e., creating a set of merged pages). Each merged page independently maintains its lineage information, i.e., keeping track of all tail records that are consolidated onto the page thus far. By maintaining explicit in-page lineage information, the current state of each page can be determined independently, and the base page can be brought up to any desired snapshot. Tail pages that are already merged and fall outside the snapshot boundaries of all active queries are called historic tail-pages. These pages are re-organized, so that different versions of a record are stored contiguously in-lined. Delta-compression is applied across different versions of tail records, and tail records are ordered based on the RIDs of their corresponding base records. Below, we describe the unique design and algorithmic features of L-Store that enables efficient transactional processing without performance deterioration of analytical processing; thereby, achieving a real-time OLTP and OLAP.

2.2 Lineage-based Storage Architecture

In L-Store, the storage layout is natively columnar and applies equally to both base and tail pages. A detailed view of our lineage-based storage architecture is presented in Figure 2. In general, one can perceive tail pages as directly mirroring the structure and the schema of base pages. As we pointed out earlier, conceptually for every record, we distinguish between base vs. tail records, where each record is assigned a unique RID. But it is important to note that the RID assigned to a base record is stable and remains constant throughout the entire life-cycle of a record, and all indexes only reference base records (base RIDs); consequently, eliminating index maintenance problem associated when update operation results in creation of a new version of the record [33, 34]. When a reader performing index lookup, it always lands at a base record, and from the base record it can reach any desired version of the record by following the table-embedded indirection to access the latest (if the base record is out-of-date) or an earlier version of the record. However, when a record is updated, a new version is

¹Fundamentally, there is no difference between base vs. tail record, the distinction is made only to ease the exposition.

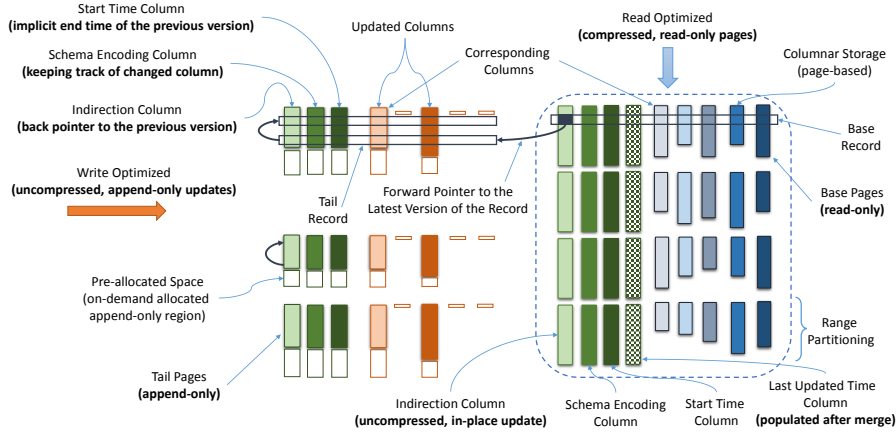


Figure 2: Detailed, unfolded view of lineage-based storage architecture (a multi-version, columnar storage model).

created. Thus, a new tail record is created to hold the new version, and the new tail record is assigned a new tail RID that is referenced by the base record (as demonstrated in Figure 2).

Each table in addition to having the standard data columns has several meta-data columns. These meta-data columns include the *Indirection* column, the *Schema Encoding* column, the *Start Time* column, and the *Last Updated Time* column. An example of table schema is shown in Table 1.

The *Indirection* column exists in both the base and tail records. For base records, the *Indirection* column is interpreted as a forward pointer to the latest version of a record residing in tail pages, essentially storing the RID of the latest version of a record. If a record has never been updated, then the *Indirection* column will hold a null value. In contrast, for tail records, the *Indirection* column is used to store a backward pointer to the last updated version of a record in tail pages. If no earlier version exists, then the *Indirection* column will point to the RID of the base record.

The *Schema Encoding* column stores the bitmap representation of the state of the data columns for each record, where there is one bit assigned for every column in the schema (excluding the meta-data columns), and if a column is updated, its corresponding bit in the *Schema Encoding* column is set to 1, otherwise is set to 0. The schema encoding enables to quickly determine if a column has ever been updated or not for base records. In tail records, the encoding tracks which columns have been updated and have explicit values as opposed to those columns that have not been updated and have an implicit special null values (denoted by \emptyset). An example of *Schema Encoding* column is provided in Table 1.

The *Start Time* column stores the time at which a base record was first installed in base pages (the original insertion time), and for a tail record, the *Start Time* column holds the time at which the record was updated, which is also the implicit end time of the previous version of the record. The *Start Time* column is essential for distinguishing between different version of the record. In addition, to the *Start Time* column, for base records, we maintain an optional *Last Updated Time* column, which is only populated after the merge process is taken place and reflects the *Start Time* of those tail records included in merged pages. Also note that the initial *Start Time* column for base records is always preserved even after the merge process for faster pruning of those records that are not visible to readers because they fall outside the reader’s snapshot. Lastly, we may add the *Base RID* column optionally to tail records to store the RIDs of their corresponding base records; this is utilized to improve the merge process. *Base RID* is a highly compressible column that would require at most two bytes when restricting the range partitioning of records to 2^{16} records.

3 FINE-GRAINED MANIPULATION

The transaction processing can be viewed as two major challenges: (1) how data is physically manipulated at the storage layer and how changes are propagated to indexes and (2) how multiple

RID	Indirection	Schema Encoding	Start Time	Key	A	B	C
Partitioned base records for the key range of k_1 to k_3							
b_1	t_8	0000	10:02	k_1	a_1	b_1	c_1
b_2	t_5	0101	13:04	k_2	a_2	b_2	c_2
b_3	t_7	0001	15:05	k_3	a_3	b_3	c_3
Partitioned base records for the key range of k_4 to k_6							
b_4	\perp	0000	16:20	k_4	a_4	b_4	c_4
b_5	\perp	0000	17:21	k_5	a_5	b_5	c_5
b_6	\perp	0000	18:02	k_6	a_6	b_6	c_6
Partitioned tail records for the key range of k_1 to k_3							
t_1	b_2	0100*	13:04	\emptyset	a_2	\emptyset	\emptyset
t_2	t_1	0100	19:21	\emptyset	a_{21}	\emptyset	\emptyset
t_3	t_2	0100	19:24	\emptyset	a_{22}	\emptyset	\emptyset
t_4	t_3	0001*	13:04	\emptyset	\emptyset	\emptyset	c_2
t_5	t_4	0101	19:25	\emptyset	a_{22}	\emptyset	c_{21}
t_6	b_3	0001*	15:05	\emptyset	\emptyset	\emptyset	c_3
t_7	t_6	0001	19:45	\emptyset	\emptyset	\emptyset	c_{31}
t_8	b_1	0000	20:15	\emptyset	\emptyset	\emptyset	\emptyset

Table 1: An example of the update and delete procedures (conceptual tabular representation).

transactions (where each transaction consists of many statements) can concurrently coordinate reading and writing of the shared data. The focus of this paper is on the former challenge, and we defer the latter to our discussion on the employed low-level synchronization and concurrency control in Section 5.

Without loss of generality, from the perspective of the storage layer, we focus on how to handle a single point update or delete in L-Store (but note that we support multi-statement transactions through L-Store’s transaction layer as demonstrated by our evaluation). Furthermore, in our technical report [31], we discuss how our model can easily be extended to deal with insertion as well. Each update may affect a single or multiple records. Since records are (virtually) partitioned into a set of disjoint ranges as shown in Table 1, each updated record naturally falls within only one range. Now for each range of records, upon the first update to that range, a set of tail pages are created (and persisted on disk optionally) for the updated columns and are added to the page directory, i.e., lazy tail-page allocation. Consequently, updates for each record range are appended to their corresponding tail pages of the updated columns only; thereby, retraining all versions of the record, avoiding in-place updates of modified data columns, and clustering updates for a range of records within their corresponding tail pages.

To describe the update procedure in L-Store, we rely on our running example shown in Table 1. When a transaction updates any column of a record for the first time, two new tail records (each tail record is assigned a unique RID) are created and appended to the corresponding tail pages. For example, consider updating the column A of the record with the key k_2 (referenced by the RID b_2) in Table 1. The first tail record, referenced by the RID t_1 , contains the original value of the updated column, i.e., a_2 , whereas implicit null values (\emptyset) are preassigned for remaining unchanged columns. Taking a snapshot of the original changed values becomes essential in order to ensure contention-free merging as discussed in Section 4.1. The second tail record contains the newly updated value for column A, namely, a_{21} , and again

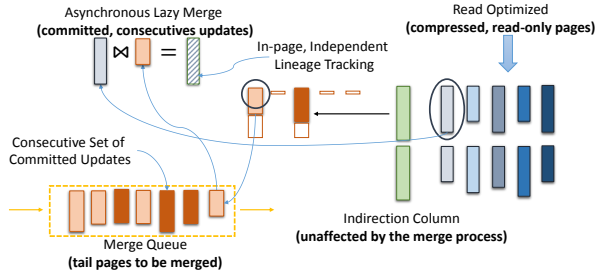


Figure 3: Lazily, independently merging of tail & base pages.

implicit special null values for the rest of the columns; a column that has never been updated does not even have to be materialized with special null values. However, for any subsequent updates, only one tail record is created, e.g., the tail record t_3 is appended as a result of updating the column A from a_{21} to a_{22} for the record b_2 .

In general, updates could either be cumulative or non-cumulative. The cumulative property implies that when creating a new tail record, the new record will contain the latest values for all of the updated columns thus far. For example, consider updating the column C for the record b_2 . Since the column C of the record b_2 is being updated for the first time, we first take a snapshot of its old value as captured by the tail record t_4 . Now for the cumulative update, a new tail record is appended that repeats the previously updated column A , as demonstrated by the tail record t_5 . If non-cumulative update approach was employed, then the tail record would consist of only the changed value for column C and not A . It is important to note that cumulation of updates can be reset at anytime. In the absence of cumulation, readers are simply forced to walk back the chain of recent versions to retrieve the latest values of all desired columns. Thus, cumulative update is an optimization that is intended to improve the read performance.

As part of the update routine, the embedded *Indirection* column (forward pointers) for base records is also updated to point to the newly created tail record. In our running example, the *Indirection* column of the record b_2 points to the tail record t_5 . Also after updating the column C of the record b_3 , the *Indirection* column points to the latest version of b_3 , which is given by t_7 . Likewise, the *Indirection* column in the tail records point to the previous version of the record. It is important to note that the *Indirection* column of base records is the only column that requires an in-place update in our architecture. However, as discussed in our low-level synchronization protocol (cf. Section 5), this is a special column that lends itself to latch-free synchronization.

Furthermore, indexes always point to base records (i.e., base RIDs), and they never directly point to any tail records (i.e., tail RIDs) in order to avoid the index maintenance cost that arise in the absence of in-place update mechanism [33]. Therefore, when a new version of a record is created (i.e., a new tail record), first, all indexes defined on unaffected columns do not have to be modified and, second, only the affected indexes are modified with the updated values, but they continue to point to base records and not the newly created tail records. Suppose there is an index defined on the column C (cf. Table 1). Now after modifying the record b_2 from c_2 to c_{21} , we add the new entry (c_{21}, b_2) to the index on the column C .² Subsequently, when a reader looks up the value c_{21} from the index, it always arrives at the base record b_2 initially, then the reader must determine the visible version of b_2 (by following the indirection if necessary) and must check if

²Optionally the old value (c_2, b_2) could be removed from the index; however, its removal may affect those queries that are using indexes to compute answers under snapshot semantics. Therefore, we advocate deferring the removal of changed values from indexes until the changed entries fall outside the snapshot of all relevant active queries.

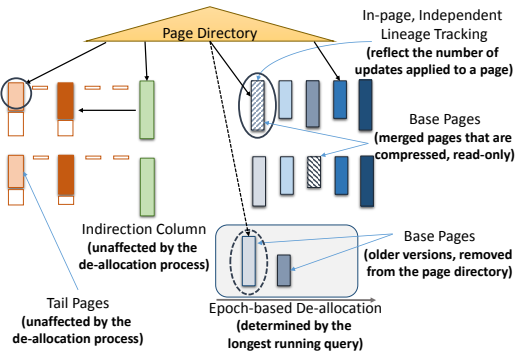


Figure 4: Epoch-based, contention-free page de-allocation.

the visible version has the value c_{21} for the column C , essentially re-evaluating the query predicates.

There are two other meta-data columns that are affected by the update procedure. The *Start Time* column for tail records simply holds the time at which the record was updated (an implicit end of the previous version). For example, the record t_7 has a start time of 19:45, which is also the implied end time of the first version of the record b_3 . The *Schema Encoding* column is a concise representation that shows which data columns have been updated thus far. For example, the *Schema Encoding* of the tail record t_7 is set to "0001", which implies that only the column C has been changed. To distinguish between whether a tail record is holding new values or it is the snapshot of old values, we add a flag to the *Schema Encoding* column, which is shown as an asterisk. For example, the tail record t_6 stores the old value of the column C , which is why its *Schema Encoding* is set to "0001*". The *Schema Encoding* can also be maintained optionally for base records as part of the update process or it could be populated only during the merge process.

Notably, when there are multiple individual updates to the same record by the same transaction, each update is written as a separate entry to tail pages. Each update results in a creation of a new tail record and only the final update becomes visible to other transactions. The prior entries are implicitly invalidated and skipped by readers. Also delete operation is simply translated into an update operation, in which all data columns are implicitly set to \emptyset , e.g., deleting the record b_1 results in creating the tail record t_8 . An alternative design for delete is to create a tail record that holds a complete snapshot of the latest version of the deleted record.

4 REAL-TIME STORAGE ADAPTION

To ensure a near optimal storage layout, outdated base pages are merged lazily with their corresponding tail pages in order to preserve the efficiency of analytical query processing. Recall that the base pages are read-only and compressed (read optimized) while the tail pages are uncompressed³ that grow using a strictly append-only technique (write optimized). Therefore, it is necessary to transform the recent committed updates accumulated in tail pages that are write optimized into read optimized form. A distinguishing feature of our lineage-based architecture is to introduce a contention-free merging process that is carried out completely in the background without interfering with foreground transactions. Furthermore, the contention-free merging procedure is applied only to the updated columns of the affected update ranges. There is even no dependency among columns during the merge; thus, the different columns of the same record can be merged completely independent of each other at different points in time. This is achieved by independently maintaining in-page lineage information for each merged page. The merge process is conceptually depicted in Figure 3, in which writer threads (i.e., update transactions) place candidate tail pages to be merged into

³Even though compression techniques such as local and global dictionaries can be employed in tail pages, but these directions are outside the scope of the current work.

the merge queue while the merge thread continuously takes pages from the queue and processes them.

4.1 Contention-free, Relaxed Merge

In L-Store, we abide to one *main design principle* for ensuring contention-free processing that is “*always operating on stable data*”. The inputs to the merge process are (1) a set of base pages (committed base records) that are read-only,⁴ thus, stable data and (2) a set of consecutive committed tail records in tail pages,⁵ thus, also stable data. The output of the merge process (that is also relaxed) is a set of newly consolidated base pages (also referred to as merged pages) with in-page lineage information that are read-only, compressed, and almost up-to-date, thus, stable data. To decouple users’ transactions (writers) from the merge process, we also ensure that the write path of the ongoing transactions does not overlap with the write path of the merge process. Writers append new uncommitted tail records to tail pages, but as stated before uncommitted records do not participate in the merge. Writers also perform in-place update of the *Indirection* column within base records to point to the latest version of the updated records in tail pages, but the *Indirection* column is not modified by the merge process. In contrast, the write path of the merge process consists of creating only a new set of read-only base pages.

Merge Algorithm The details of the merge algorithm, conceptually resembling the standard left-outer join, consists of (1) identifying a set of committed tail records in tail pages; (2) loading the corresponding outdated base pages; (3) consolidating the base and tail pages while maintaining the in-page lineage; (4) updating the page directory; and (5) de-allocating the outdated base pages. The pseudo code for the merge is shown in Algorithm 1, where each of the five mentioned steps are also highlighted.

Step 1: Identify committed tail records in tail pages: Select a set of consecutive fully committed tail records (or pages) since the last merge within each update range.

Step 2: Load the corresponding outdated base pages: For a selected set of committed tail records, load the corresponding outdated base pages for the given update range (limit the load to only outdated columns). This step can further be optimized by avoiding to load sub-ranges of records that have not yet changed since the last merge. No latching is required when loading the base pages.

Step 3: Consolidate the base and tail pages: For every updated column, the merge process will read n outdated base pages and applies a set of recent committed updates from the tail pages and writes out m new pages.⁶ First the Base RID column of the committed tail pages (from Step 1) are scanned in reverse order to find the list of the latest version of every updated record since the last merge (a temporary hashtable may be used to keep track whether the latest version of a record is seen or not). Subsequently, applying the latest tail records in a reverse order to the base records until an update to every record in the base range is seen or the list is exhausted, skipping any intermediate versions for which a newer update exists in the selected tail records. If a latest tail record indicates the deletion of the record, then the deleted record will be included in the consolidated records. The merged pages will keep track of the lineage information in-page, i.e., tracking

⁴The *Indirection* column is the only column that undergoes in-place update that also never participates in the merge process.

⁵Note that not every committed update has to be applied as the merge process is relaxed, and the merge eventually process all committed tail records.

⁶At most up to one merged page per column could be left underutilized for a range of records after the merge process. To further reduce the underutilized merged pages, one may define finer range partitioning for updates (e.g., 2^{12} records), but operate merges at coarser granularity (e.g., 2^{16} records). This will provide the benefit of locality of access for readers given smaller range size of 2^{12} , yet it provides a better space utilization and compression for newly created merge pages when larger ranges are chosen (cf. Section 4.3).

Algorithm 1: Merge Algorithm

```

Input      : Queue of unmerged committed tail pages (mergeQ)
Output    : Queue of outdated and consolidated base pages to be deallocated
              (deallocateQ)

1 while true do
2   // Step 1
3   // wait until the the concurrent merge queue is not empty
4   if mergeQ is not empty then
5     // Step 2
6     // fetch references to a set of committed tail pages
7     batchTailPage ← mergeQ.dequeue()
8     // create a copy of corresponding base pages
9     batchConsPage ← batchTailPage.getBasePageCopy()
10    decompress(batchConsPage)
11    // track if it has seen the latest update of every record
12    HashMap seenUpdatesH
13    //reading a set of tail pages in reverse order
14    // Step 3
15    for i = 0; i < batchTailPage.size; i ← i + 1 do
16      tailPage ← batchTailPages[i]
17      for j = k - 1; j ≥ tailPage.size; j ← j - 1 do
18        record[j] ← jth record in the tailPage
19        RID ← record[j].RID
20        if seenUpdatesH does not contain RID then
21          seenUpdatesH.add(RID)
22          // copy the latest version of record into consolidated pages
23          batchConsPage.update(RID, record[j])
24        end
25        if all RIDs OR all tail pages are seen then
26          compress(batchConsPage)
27          persist(batchConsPage)
28          stop examining remaining tail pages
29        end
30      end
31    end
32    // Step 4
33    // fetch references to the corresponding base pages
34    batchBasePage ← batchTailPage.getBasePageRef()
35    // update page directory to point to the consolidated base pages
36    PageDirect.swap(batchBasePage, batchConsPage)
37    // Step 5
38    // queue outdated pages for deallocation once readers prior merge are drained
39    deallocateQ.enqueue(batchBasePage)
40  end
41 end

```

how many tail records have been consolidated thus far. Any compression algorithm (e.g., dictionary encoding) can be applied on the consolidated pages (on column basis) followed by writing the compressed pages into newly created pages. Moreover, the old Start Time column is remained intact during the merge process because this column is needed to hold the original insertion time of the record.⁷ Therefore, to keep track of the time for the consolidated records, the Last Updated Time column is populated to store the Start Time of the applied tail records. The Schema Encoding column may also be populated during the merge to reflect all the columns that have been changed for each record.

Step 4: Update the page directory: The pointers in the page directory are updated to point to the newly created merged pages. Essentially this is the only foreground action taken by the merge process, which is simply to swap and update pointers in the page directory – an index structure that is updated rarely only when new pages are allocated.

Step 5: De-allocate the outdated base pages: The outdated base pages are de-allocated once the current readers are drained naturally via an epoch-based approach. The epoch is defined as a time window, in which the outdated base pages must be kept around as long as there is an active query that started before the merge process. Pointers to the outdated base pages are kept in a queue to be re-claimed at the end of the query-driven epoch-window. The pointer swapping and the page de-allocation are illustrated in Figure 4. ■

⁷The Start Time column is also highly compressible column with a negligible space overhead to maintain it.

RID	Indirection	Schema Encoding	Start Time	Last Updated Time	Key	A	B	C
Partitioned base records for the key range of k_1 to k_3 ; Tail-page Sequence Number (TPS) = 0								
b_1	t_8	0000	10:02		k_1	a_1	b_1	c_1
b_2	t_5	0101	13:04		k_2	a_2	b_2	c_2
b_3	t_7	0001	15:05		k_3	a_3	b_3	c_3
Relevant tail records (below TPS $\leq t_7$ high-watermark) for the key range of k_1 to k_3								
t_5	t_4	0101	19:25		\emptyset	a_{22}	\emptyset	c_{21}
t_7	t_6	0001	19:45		\emptyset	\emptyset	\emptyset	c_{31}
Resulting merged records for the key range of k_1 to k_3 ; TPS = t_7								
b_1	t_8	0000	10:02	10:02	k_1	a_1	b_1	c_1
b_2	t_5	0101	13:04	19:25	k_2	a_{22}	b_2	c_{21}
b_3	t_7	0001	15:05	19:45	k_3	a_3	b_3	c_{31}

Table 2: An example of the relaxed and almost up-to-date merge procedure (conceptual tabular representation).

An example of our merge process is shown in Table 2 based on our earlier update example, in which we consolidate the first seven tail records (denoted by t_1 to t_7) with their corresponding base pages. The resulting merged pages are shown, where the affected records are highlighted. Note that only the updated columns are affected by the merge process (and the *Indirection* column is not affected). Furthermore, not all updates are needed to be applied, only the latest version of every updated record needs to be consolidated while the other entries are simply discarded. In our example, only the tail records t_5 and t_7 participated in the merge, and the rest were discarded.

Merge Correctness Analysis A key distinguishing feature of our lineage-based storage architecture is to allow contention-free merging of tail and base pages without interfering with concurrent transactions. To formalize our merge process, we prove that merge operates only on stable data while maintaining in-page lineage without any information loss and that the merge does not limit users’ transactions to access and/or modify the data that is being merged.

LEMMA 4.1. *Merge operates strictly on stable data.*

PROOF. By construction, we enforced that merge “always operate on stable data”. The inputs to the merge process are (1) a set of base pages consisting of committed base records that are read-only, i.e., stable data and (2) a set of consecutive committed tail records in tail pages, thus, also stable data. The output of the merge process is a set of newly merged pages that are read-only, i.e., stable data as well. Hence, the merge process strictly takes as inputs stable data and produces stable data as well. \square

LEMMA 4.2. *Merge safely discards outdated base pages without violating any query’s snapshot.*

PROOF. In order to support snapshot isolation semantics and time travel queries, we need to ensure that earlier versions of records that participate in the merge process are retained. Since we never perform in-place updates and each update is transformed into appending a new version of the record to tail pages, then as long as tail pages are not removed, we can ensure that we have access to every updated version. But recall that outdated base pages are de-allocated using our proposed epoch-based approach after being merged. Also note that base pages contain the original values of when a record was first created. Therefore, any original values that later were updated must be stored before discarding outdated base pages after a merge is taken place. In another words, we must ensure that outdated base pages are discarded safely.

As a result, the two fundamental criteria, namely, relaxing the merge (i.e. constructing an almost up-to-date snapshot) and operating on stable data, are not sufficient to ensure the *safety property* of the merge. The last missing piece that enables safety of the merge is accomplished by taking a snapshot of the original values when a column is being updated for the first time (as described in Section 3). In other words, we have further strengthened our *data stability* criterion by ensuring even *stability in the committed history*. Hence, outdated base pages can be safely discarded without any information loss, namely, the merge process is safe. \square

RID	Indirection	Schema Encoding	Start Time	Last Updated Time	Key	A	B	C
Recently merged records for the key range of k_1 to k_3 ; TPS = t_7								
b_1	t_8	0000	10:02	10:02	k_1	a_1	b_1	c_1
b_2	t_{12}	0101	13:04	19:25	k_2	a_{22}	b_2	c_{21}
b_3	t_{11}	0001	15:05	19:45	k_3	a_3	b_3	c_{31}
Partitioned tail records for the key range of k_1 to k_3								
t_1	b_2	0100*	13:04		\emptyset	a_2	\emptyset	\emptyset
t_2	t_1	0100	19:21		\emptyset	a_{21}	\emptyset	\emptyset
t_3	t_2	0100	19:24		\emptyset	a_{22}	\emptyset	\emptyset
t_4	t_3	0001*	13:04		\emptyset	\emptyset	\emptyset	c_2
t_5	t_4	0101	19:25		\emptyset	a_{22}	\emptyset	c_{21}
t_6	b_3	0001*	15:05		\emptyset	\emptyset	\emptyset	c_3
t_7	t_6	0001	19:45		\emptyset	\emptyset	\emptyset	c_{31}
t_8	b_1	0000	20:15		\emptyset	\emptyset	\emptyset	\emptyset
t_9	t_5	0010*	13:04		\emptyset	\emptyset	b_2	\emptyset
t_{10}	t_9	0010	21:25		\emptyset	\emptyset	b_{21}	\emptyset
t_{11}	t_7	0001	21:30		\emptyset	\emptyset	\emptyset	c_{32}
t_{12}	t_{10}	0110	21:55		\emptyset	a_{23}	b_{21}	\emptyset

Table 3: An example of the indirection interpretation and lineage tracking (conceptual tabular representation).

THEOREM 4.3. *The merge process and users’ transactions do not contend for base and tail pages or the resulting merged pages, namely, the merge process is contention-free.*

PROOF. As part of ensuring contention-free merge, we have already shown that merge operates on stable data (proven by Lemma 4.1) and that there is no information loss as a result of the merge process (proven by Lemma 4.2). Next we prove that the write path of the merge process does not overlap with the write path of users’ transactions (i.e., writers). Recall that writers append new uncommitted tail records to tail pages, but as stated before, uncommitted records do not participate in the merge. Writers also perform in-place update of the *Indirection* column within base records to point to the latest version of the updated records in tail pages, but the *Indirection* column is not modified by the merge process. In contrast, the write path of the merge process consists of creating only a new set of read-only merged pages and eventually discarding the outdated base pages safely.

Therefore, we must show that safely discarding base pages does not interfere with users’ transactions. In particular, as explained in Lemma 4.2, if the original values were not written to tail records at the time of the update, then during the merge process, we were forced to store them somewhere or encounter information loss. It is not even clear where would be the optimal location for storing the original values. A simple minded approach of just adding them to tail pages would have broken the linear order of changes to records such that the older values would have appeared after the newer values, and it would have interfered with the ongoing update transactions. But, more importantly, the need to store the old values at any location would have implied that during the merge process multiple coordinated actions were required to ensure consistency across modification to isolated locations; hence, breaking the contention-free property of the merge. Therefore, by storing the original updated values at the time of update, we trivially eliminate all the potential contention during the merge process in order to safely discarding outdated base pages.

As a result, users’ transactions are completely decoupled from the merge process, and users’ transactions and the merge process do not contend over base, tail, or merged pages. \square

4.2 Maintaining In-Page Lineage

The lineage of each base page and consequently merged pages is maintained within each page independently as a result of the merge process. In-page lineage information is instrumental to decouple the merge and update operations and to allow independent merging of the different columns of the same record at different points in time. In-page lineage information is captured using a rather simple and elegant concept, which we refer to as *tail-page sequence number (TPS)* in order to keep track of how many updated entries (i.e., tail records) from tail pages have been applied to their corresponding base pages after a completion of a merge.

Original base pages always start with TPS set to 0, a value that is monotonically increasing after every merge. Again to ensure this monotonicity property, as stressed earlier, always a consecutive set of committed tail records are used in the merge process.

TPS is also used to interpret the indirection pointer (also a monotonically increasing value) by readers after the merge is taken place. Consider our running example in Table 2. After the first merge process, the newly merged pages have TPS set to 7, which implies that the first seven updates (tail records t_1 to t_7) in the tail pages have been applied to the merged pages. Consider the record b_2 in the base pages that has an indirection value pointing to t_5 (cf. Table 2), there are two possible interpretations. If the transaction is reading the base pages with TPS set to 0, then the 5^{th} update has not yet reflected on the base page. Otherwise if the transaction is reading the base pages with TPS 7, then the update referenced by indirection value t_5 has already been applied to the base pages as seen in Table 2. Notably, the *Indirection* column is updated only in-place (also a monotonically increasing value) by writers, while merging tail pages does not affect the indirection value.

More importantly, we can leverage the TPS concept to ensure read consistency of users' transactions when the merge is performed lazily and independently for the different columns of the same records. Therefore, when the merge of columns is decoupled, each merge occurs independently and at different points in time. Consequently, not all base pages are brought forward in time simultaneously. Additionally, even if the merge occurs for all columns simultaneously, it is still possible that a reader reads base pages for the column A before the merge (or during the merge before the page directory is updated) while the same reader reads the column C after the merge; thus, reading a set of inconsistent base and merged pages.

LEMMA 4.4. *An inconsistent read with concurrent merge is always detectable.*

PROOF. Since each base page independently tracks its lineage, i.e., its TPS counter; therefore, TPS can be used to verify the read consistency. In particular, for a range of records, all read base pages must have an identical TPS counter; otherwise, the read will be inconsistent. Hence, an inconsistent read across different columns of the same record is always detectable. \square

THEOREM 4.5. *Constructing consistent snapshots with concurrent merge is always possible.*

PROOF. As proved in Lemma 4.4, the read inconsistency is always detectable. Furthermore, once a read inconsistency is encountered, then each page is simply brought to the desired query snapshot independently by examining its TPS and the indirection value and consulting the corresponding tail pages using the logic outlined earlier. Hence, consistent reads by constructing consistent snapshots across different columns of the same record is always possible. \square

TPS, or an alternative but similar counter conceptually, could be used as a high-water mark for resetting the cumulative updates as well. Continuing with our running scenario, in which we have the original base pages with the TPS 0 (as shown in Table 2), the merged pages with the TPS 7 (as shown in Table 3). For simplicity, we assume the cumulation was also reset after the 7^{th} tail record. For the record b_2 , we see that the indirection pointer is t_{12} , for which we know that the cumulative update has been reset after the 7^{th} update. This means that the tail record t_{12} does not carry updates that were accumulated between tail records 1 to 7. Suppose that the record was updated four times, where the update entries in the tail pages are 3^{rd} , 5^{th} , 10^{th} , and 12^{th} tail records. The tail record t_5 is a cumulative and carries the updated

values from the tail record t_3 . However, the tail record t_{10} is not cumulative (reset occurred at the 8^{th} update), whereas the tail record t_{12} is cumulative, but carries updates only from the tail record t_{10} and not from t_5 and t_3 . Suppose that a transaction is reading the base pages with the TPS 0, then to reconstruct the full version of the record b_2 , it must read both the tail records t_5 and t_{12} (while skipping 3^{rd} and 10^{th}). But if a transaction is reading from the merged pages with the TPS 7, then it is sufficient to only read the tail record t_{12} to fully reconstruct the record because the 3^{rd} and 5^{th} updates have already been applied to the merged pages.

4.3 Record Partitioning Trade-offs

When choosing the range of records for partitioning (i.e., update range) there are several dimensions that needs to be examined. An important observation is that regardless of the range size, recent updates to tail pages will be memory resident and no random disk I/O is required. This trend is supported by continued increase in the size of main memory and the fact that the entire OLTP database is expected to fit in the main memory [9, 17].

In our evaluation, we did an in-depth study of the impact of the range size, and we observed that the key deciding factor is the frequency at which the merges are processed. How frequent a merge is initiated is proportional to how many tail records are accumulated before the merge process is triggered. We further experimentally observed that the update range sizes in the order of 2^{12} to 2^{16} exhibit a superior overall performance vs. data fragmentation depending on the workload update distribution. Because for a smaller update range size, we may have many corresponding half-filled tail pages, but as the range size increases, the cost of half-filled tail pages are amortized over a much larger set of records.⁸ Furthermore, the range size affects the clustering of updates in tail pages. For larger the range size, it is more likely that cache misses occur when scanning the recent update that are not merged yet. Again, considering that recent cache sizes are in order of tens of megabytes, the choice of any range value between 2^{12} to 2^{16} is further supported. As noted before, one may choose a finer range partitioning for handling updates (i.e., update range), e.g., 2^{12} , to improve locality of access while choosing coarser virtual range sizes when performing merges, essentially forcing the merge to take-in as input a set of consecutive update ranges that have been updated, e.g., choosing 2^4 consecutive 2^{12} ranges in order to merge $2^{12} \times 2^4 = 2^{16}$ records.

For example, suppose the scan operation (even if there are concurrent scans) may access 2 columns, assume each column is 2^3 bytes long. We further assume that the merge can keep up, namely, even for 2^{16} update range size, the number of tail records yet to be merged is less than 2^{16} (as shown in Section 6, such merging rate can be achieved while executing up to 16 concurrent update transactions). The overall scan footprint (combining both base pages and tail pages) is approximately $2^{16} \times 2^3 \times 2 \times 2 = 2^{21}$ (2 MB), which certainly fits in today's processor cache (in our evaluation, we used Intel Xeon E5-2430 processor, which has 15 MB cache size). Thus, even as scanning base records, if one is forced to perform random lookup within a range of 2^{16} tail records, the number of cache misses are limited compared to when the range size was beyond the cache capacity.

Another criteria for selecting an effective update range size is the need for RID allocation. In L-Store, upon the first update to a range of records (e.g., 2^{12} to 2^{16} range), we pre-allocate 2^{12} to 2^{16} unused RIDs for referencing its corresponding tail pages. Tail RIDs are special in a sense they are not added to indexes and no unique constraint is applied on them. Once the tail RID range is

⁸To reduce space under-utilization, tail pages could be smaller than base pages, for instance, tail pages could be 4 KB while base pages are 32 KB or larger.

fully used, then either a new unused RID range is allocated or an existing underutilized tail RID range can be re-assigned (partially used RID range must satisfy TPS monotonicity requirement). Furthermore, in order to avoid overlapping the base and tail RIDs, one could assign tail RIDs in the reverse order starting from 2^{64} ; therefore, tail RIDs will be monotonically decreasing, and the TPS logic must be reversed accordingly. The benefit of reverse assignment is that while scanning page directory for base pages, there is no need to first read and later skip tail page entries (read optimization).

5 FAST TRANSACTIONAL CAPABILITIES

In order to support concurrent transactions where each transaction may consist of many statements, any database engine must provide necessary functionalities to ensure the correctness of concurrent reads and writes of the shared data. Furthermore, transaction logging is required in order to recover the system from crash and media failure. In this section, we focus on low-level synchronization protocol and logging requirements. In terms of concurrency protocol for transaction processing, any existing protocols can be leveraged because L-Store primarily focuses on the storage architecture. In particular, we relied on our recently proposed optimistic concurrency model in [32] that supports full ACID properties for multi-statement transactions, and we also employed the speculative reads proposed in [18]. The details of the concurrency protocol is presented in our technical report [31].

Low-level Synchronization Protocol In terms of low-level latching, our lineage-based storage has a set of unique benefits, namely, readers do not have to latch the read-only base pages or fully committed tail pages. Also there is no need to latch partially committed tail pages when accessing committed records. More importantly, writers never modify base pages (except the *Indirection* column) nor the fully committed tail pages, so no latching is required for stable pages. The *Indirection* column is at most 8-byte long; therefore, writers can simply rely on atomic compare-and-swap (CAS) operators to avoid latching the page.

As part of the merge process, no latching of tail and base pages are required because they are not modified. The only latching requirement for the merge is updating the page directory to point to the newly created merged pages. Therefore, every affected page in the page directory are latched one at a time to perform the pointer swap or alternatively atomic CAS operator is employed for each entry (pointer swap) in the page directory. Alternatively, the page directory can be implemented using latch-free index structures such as Bw-Tree [20].

Recovery and Logging Protocol Our lineage-based storage architecture consists of read-only base pages (that are not modified) and append-only updates to tail pages (which are not modified once written). When a record is updated, no logging is required for base pages (because they are read-only), but the modified tail pages requires redo logging. Again, since we eliminate any in-place update for tail pages, no undo log is required. Upon a crash, the redo log for tail pages are replayed, and for any uncommitted transactions (or partial rollback), the tail record is marked as invalid (e.g., tombstone), but the space is not reclaimed until the compression phase.

The one exception to above rule for logging and recovery is the *Indirection* column, which is updated in-place. There are two possible recovery options: (1) one can rely on standard undo-redo log for the *Indirection* column only or (2) one can simply rebuild the *Indirection* column upon crash. The former option can further be optimized based on the realization that tail pages undergo strictly redo policy and aborted transactions do not physically remove the aborted tail records as they are only marked as tombstones. Therefore, it is acceptable for the *Indirection* column to continue pointing to tombstones, and from the tombstones finding the latest

committed values. As a result, even for the *Indirection* column only the redo log is necessary. For the latter recovery option, as discussed earlier, to speedup the merge process, we materialize the *Base RID* column in tail records that can be used to populate the *Indirection* column after the crash. Alternatively, even without materializing an additional RID column, one can follow back-pointers in the *Indirection* column of tail records to fetch the base RID because the very first tail record always points back to the original base record.

The merge process is idempotent because it operates strictly on committed data and repeated executions of the merge always produce the exact same results given a set of base pages, their corresponding tail pages, and a merge threshold that dictates how many consecutive committed tail records to be used in the merge process. Therefore, only operational logging is required for the merge process. Also updating the entries in the page directory upon completion of the merge process simply requires standard index logging (both undo-redo logs). If crash occurs during the merge, simply the partial merge results can be ignored and the merge can be restarted.

6 EXPERIMENTAL EVALUATION

In order to study the impact of high-throughput transaction processing in the presence of long-running analytical queries, we carried out a comprehensive set of experiments. These experiments were performed using an existing micro benchmark proposed in [18, 32], i.e., a comprehensive transactional YCSB-like benchmark [8], for the sake of a fair comparison and evaluation. This benchmark allows us to study different storage architectures by narrowing down the impact of concurrency with respect to the active data set by adjusting the degree of contention between readers and writers.

6.1 Experimental Setting

We evaluate the performance of various aspects of our real-time OLTP and OLAP system. Our experiments were conducted on a two-socket Intel Xeon E5-2430 @ 2.20 GHz server that has 6 cores per socket with hyper-threading enabled (providing a total of 24 hardware threads). The system has 64 GB of memory and 15 MB of L3 cache per socket. We implemented a complete working prototype of L-Store and compared it against two different techniques, (i) In-place Update + History and (ii) Delta + Blocking Merge, which are described subsequently. The prototype was implemented in Java (using JDK 1.7). Our primary focus here is to simultaneously evaluate read and write throughputs of these systems under various transactional workloads concurrently executed with long-running analytical queries, which is the key characteristic of any real-time OLTP and OLAP system.

Our employed micro benchmark defined in [18, 32] consists of three key types of workloads: (1) low contention, where the active set is 10M records; (2) medium contention, where the active set is 100K records; and (3) high contention, where the active set is 10K records. It is important to note that the data size is not limited to the active set and can be much larger (millions or billions of records). Similar to [18, 32], we consider two classes of transactions. (1) *Read-only transactions* executed under snapshot isolation semantics that scan up to 10% of the data to model TPC-H style analytical queries. (2) *Short update transactions* executed under committed read semantics to model TPC-C and TPC-E transactions, in which each *short update transaction* consists of 8 read and 2 write statements over a table schema with 10 columns. In addition, we vary the ratio of read/writes in these update transactions to model different customer scenarios with different read/write degrees. By default, transactional throughput of these schemes are evaluated while running (at least) one scan thread and one merge thread to create the real-time OLTP

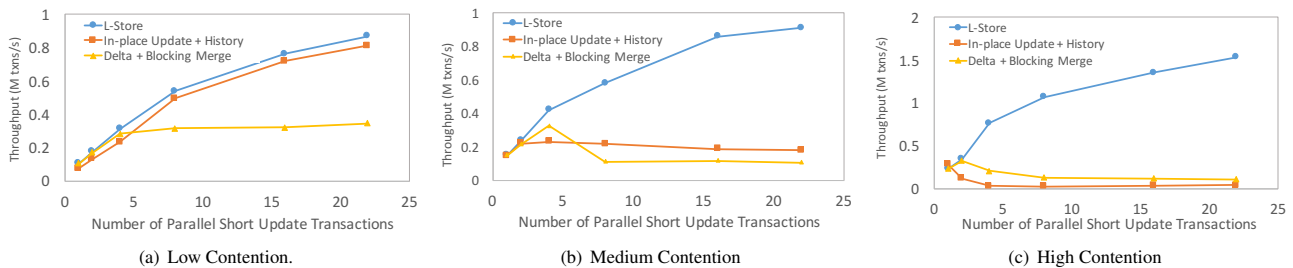


Figure 5: Scalability under varying contention level.

and OLAP scenario. Unless stated explicitly, the percentage of reads and writes in the transactional workload is fixed at 80% and 20%, respectively. On average 40% of all columns are updated by the writers. Lastly, the page size is set to 32 KB for both base and tail pages because a larger page size often results in a higher compression ratio suitable for analytical workloads [13].

Next we describe the two techniques that are compared with L-Store. We point out the primary features of these techniques and describe it with respect to L-Store. For fairness, across all techniques, we have maintained columnar storage, maintained a single primary index for fast point lookup, and employed the embedded-indirection column to efficiently access the older/newer versions of the records. Additionally, logging has been turned off for all systems as logging could easily become the main bottleneck (unless sophisticated logging mechanisms such as group commits and/or enterprise-grade SSDs are employed). In the In-place Update + History technique, we are required to write both redo-undo logs for all updates while for L-Store and Delta + Blocking Merge only redo log is needed due to their append-only scheme.

In-place Update + History (IUH): A prominent storage organization is to append old versions of records to a history table and only retain the most recent version in the main table, updating it in-place. Such table organization is motivated by commercial systems such as [26]; thus, our In-place Update + History is inspired by such table organization that avoids having multiple copies and representations of the data. However, due to the nature of the in-place update approach, each page requires standard shared and exclusive latches that are often found in major commercial big data systems. In addition to the page latching requirement, if a transaction aborts, then the update to the page in the main table is undone, and the previous record is restored. Scans are performed by constructing a consistent snapshots, namely, if records in the main table are invisible with respect to query’s read time, then the older versions of the records are fetched from the history table by following the indirection column. In our implementation of In-place Update + History, we also ignored other major costs of in-place update over the compressed data, in which the new value may not fit in-place due to compression and requires costly page splits or shifting data within the page as part of update transactions. We further optimized the history table to include only the updated columns as opposed to inserting all columns naively.

Delta + Blocking Merge (DBM): This technique is inspired by [14], where it consists of a main store and a delta store, and undergoes a periodic merging and consolidation of the main and delta stores. However, the periodic merging requires the draining of all active transactions before the merge begins and after the merge ends. Although the resulting contention of the merge appears to be limited to only the boundary of the merge for a short duration, the number of merges and the frequency at which this merge occurs has a substantial impact on the overall performance. We optimized the delta store implementation to be columnar and included only the updated columns [27]. Additionally, we applied our range partitioning scheme to the delta store by dedicating a separate delta store for each range of records to further reduce the cost of merge operation in presence of data skew. The partitioning

	L-Store	IUH	DBM
Scan Performance (in secs.)	0.24	0.28	0.38

Table 4: Scan performance for different systems.

allow us to avoid reading and writing the unchanged portion of the main store.

6.2 Experimental Results

In what follows, we present our comprehensive evaluation results in order to compare and study our proposed L-Store with respect to state-of-the-art approaches.

Scalability under contention: In this experiment, we show how transaction throughput scales as we increase the number of update transactions, in which each update transaction is assigned to one thread. For the scalability experiment, we fix the number of reads to 8 and writes to 2 for each transaction against a table with active set of $N = 10$ million rows. Figure 5(a) plots the transaction throughput (y-axis) and the number of update threads (x-axis). Under low contention, the throughput for L-Store and In-place Update + History scales almost linearly before data is spread across the two NUMA nodes. The Delta + Blocking Merge approach however does not scale beyond a small number of threads due to the draining of active transaction before/after of each merge process, which brings down the transaction throughput noticeably. With increasing number of threads, the number of merges and the draining of active transactions become more frequent, which reduces the transaction throughput significantly. The In-place Update + History approach has lower throughput compared to L-Store due to the exclusive latches held for data pages that block the readers attempting to read from the same pages. The presence of a single history table also results in reduced locality for reads and more cache misses.

In addition, we study the impact of increasing the degree of contention by varying the size of the active set. For a fixed degree of contention, we vary the number of parallel update transactions from 1 to 22. For both medium contention (Figure 5(b)) and high contention (Figure 5(c)), we observe that L-Store consistently outperforms the In-place Update + History and Delta + Blocking Merge techniques as the number of parallel transactions is increased. For medium contention, we observed a speedup of up to 5.09 \times compared to the In-place Update + History technique and up to 8.54 \times compared to the Delta + Blocking Merge technique. Similarly for high contention, we observed up to 40.56 \times and 14.51 \times speedup with respect to the In-place Update + History and Delta + Blocking Merge techniques, respectively. The greater performance gap is attributed to the fact that in In-place Update + History, latching contention on the page is increased that is altogether eliminated in L-Store. In Delta + Blocking Merge, since the active set is smaller, and all updates are concentrated to smaller regions, the merging frequency is increased, which proportionally reduces the overall throughput due to the constant draining of all active transactions. Finally, due to the smaller active set sizes in the medium- and high-contention workloads, the cache misses are also reduced as the cache-hit ratio increases. As a consequence, the transaction throughput also increases proportionately.

Scan Scalability: Scan performance is an important metric for real-time OLTP and OLAP systems because it is the basic building block for assembling complex ad-hoc queries. We measure the

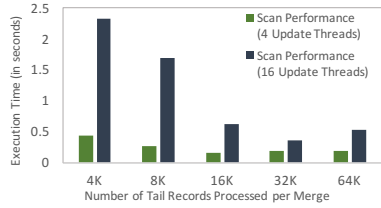


Figure 6: Scan performance.

scan performance of L-Store by computing the SUM aggregation on a column that is continuously been updated by the concurrent update transactions. Thus, the goal of this experiment is to determine whether the merge can keep up with high-throughput OLTP workloads. As such, this scenario captures the worst-case scan performance because it may be necessary for the scan thread to search for the latest values in the merged page or tail pages when the merge cannot cope with the update throughput. For columns which do not get updated, the latest values are available in the base page itself, as described before. In this experiment (Figure 6), we study the single-threaded scan performance with one dedicated merge thread. We vary the number of tail records (M) that are processed per merge (x -axis) and observe the corresponding scan execution time (y -axis) while keeping the range partitioning fixed at 64K records. We repeat this experiment by fixing the number of update threads to 4 and 16, respectively. In general, we observe that as we increase M , the scan execution time decreases. The main reasoning behind this observation is that the scan thread visits tail pages for the latest values less often because the merge is able to keep up. However, for the smaller values of M , the merge is triggered more frequently and cannot be sustained. Additionally, the overall cost of the merge is increased because the cost of merge is amortized over fewer tail records while still reading the entire range of 64K base records. Notably, if we delay the merge by accumulating too many tail records, then there is slight deterioration in performance. Therefore, it is important to balance the merge frequency vs. the amortization cost of the merge for the optimal performance, which based on our evaluation, it is when M is set to around 50% of the range size.

We also compare the single-threaded scan performance (for low contention and 4K range size) of L-Store with the other two techniques in the presence of 16 concurrent update threads (as shown in Table 4). Our technique outperforms the In-place Update + History and Delta + Blocking Merge techniques by 14.28% and 36.84%, respectively. It is important to note that smaller update range sizes, namely, assigning separate tail pages for each 4K base records instead of 64K base records, increases the overall scan performance by improving the locality of access within tail pages. Therefore, as elaborated previously in Section 4.3, it is beneficial to apply (virtual) fine-grained partitioning over base records (e.g., 4K records) to handle updates in order to improve locality of access within tail pages while applying (virtual) coarser-grained partitioning (e.g., 64K records) when performing the merge in order to reduce the space fragmentation in the resulting merged pages.

Impact of varying the workload read/write ratio: Short update transactions update only a few records. A typical transactional workload comprises of 80% read statements and 20% writes [18]. However, our goal is to explore the entire spectrum from a read-intensive workload (read/write ratio 10:0) to a write-intensive workload (read/write ratio 0:10) while fixing the number of update threads to 16. Figure 7(a) shows transaction throughput (y -axis) as the ratio of read-only transactions varies in the workload (x -axis) with low contention. As expected, the performance of all the schemes increases as we increase the ratio of reads in the transactions because contention is a function of writes. As we have more writes in the workload, In-place Update + History technique suffers from increased contention as acquiring read latches

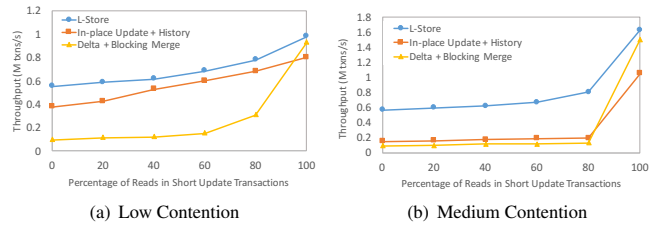


Figure 7: Impact of varying the read/write ratio of short update transactions.

conflict with the exclusive latches resulting in an extended wait time. The performance of the Delta + Blocking Merge technique also exacerbates since increasing the number of writes increases the number of merges performed. This brings down the performance further due to frequent halt of the system while draining active transactions. However, note that the gap between all of the schemes is the least when the workload consists of 100% reads. In summary, the speedup obtained with respect to In-place Update + History is up to 1.45 \times and up to 5.78 \times with respect to Delta + Blocking Merge technique. Note, even for 100% read, In-place Update + History continues to pay the cost of acquiring read latches on each page.

We repeat the same experiment but restrict the active set size to 100K rows (Figure 7(b)). L-Store significantly outperforms the other techniques across all workloads while varying the read/write ratio. But the performance gap is similar with respect to the low contention scenario when there are no update statements in the workload. The speedup obtained compared to In-place Update + History and Delta + Blocking Merge techniques is up to 4.19 \times and up to 6.34 \times respectively.

Impacts of long-read transactions: As mentioned previously, it is not uncommon to have long-running read-only transactions in real-time OLTP and OLAP systems. These analytical queries touch a substantial part of the data compared to the short update transactions, and the main goal is to reduce the interference between OLTP and OLAP workloads. In this experiment, we investigate the performance of the different schemes in the presence of these long-running read-only transactions, which on an average touch 10% of the base table. We fix the number of concurrent active transactions to 17 while increasing the number of concurrent read-only transactions from 1 to 16 (the short transactions simultaneously vary from 16 to 1). We also allocated a single merge thread for L-Store and Delta + Blocking Merge. Figures 8(a)-8(b) represent the scenario for a low contention workload, while Figures 8(c)-8(d) represent the scenario for medium contention.

We observe that for both low and medium contention, there is an increase in throughput for both long-read transactions and short update transactions when the number of threads are increased. Moreover, the performance of read-only transaction increases for the medium contention scenario for all the techniques as the updates are restricted to a small portion of the data resulting in a higher read throughput. In other words, majority of the read-only transactions touch portions of the data in which updates do not take place resulting in higher throughput. For read-only transactions, our technique outperforms Delta + Blocking Merge up to 1.97 \times and 2.37 \times for low and medium contention workloads, respectively. For short update transactions, we outperform In-place Update + History and Delta + Blocking Merge by at most 5.37 \times and 7.91 \times , respectively, for medium-contention workload. In the earlier experiments, we had demonstrated that L-Store outperforms other leading approaches for update-intensive workloads, and in this experiment, we further strengthen our claim that L-Store substantially outperforms the leading approaches in the mixed OLTP and OLAP workload as well, the latter is due to our novel contention-free merging that does not interfere with the OLTP portion of the workload.

Impacts of comparing row vs. columnar layouts We revisit the scan and point query performance while considering both row

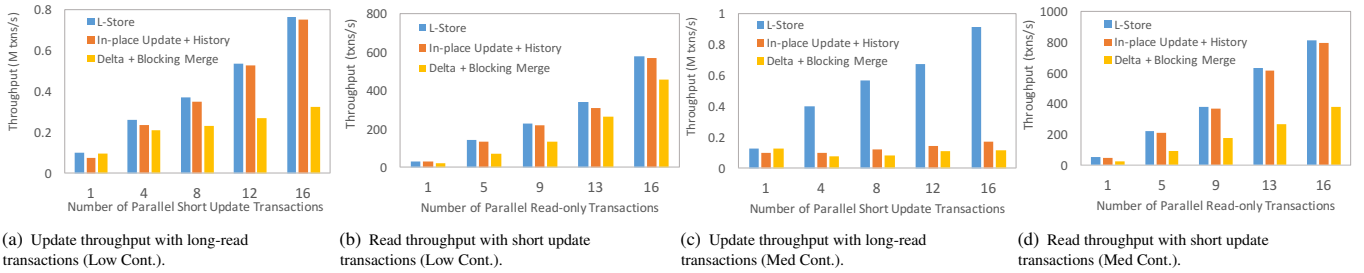


Figure 8: Impact of varying the number of short update vs. long read-only transactions.

	L-Store (Column)	L-Store (Row)
Scan Performance without updates (in secs.)	0.043	0.196
Scan Performance with updates (in secs.)	0.24	0.66

Table 5: Scan performance based on row vs. columnar layouts.

	10% of Columns	20%	40%	80%	All Columns
L-Store (Column)	1.46	1.35	1.17	1.08	0.98
L-Store (Row)	1.45	1.45	1.45	1.45	1.45

Table 6: Point query performance vs. percentage of columns read (M txns/second).

	Lineage-based	Log-structured
Update throughput (16 threads) with 1 scan thread	0.76M txns/sec	0.15M txns/sec
Scan performance (in sec) with update txns (16 threads)	0.24	0.63

Table 7: Update/Scan performance of lineage-based vs. log-structured storage architecture.

and columnar storage layouts. To enable this comparison, we additionally developed a variation of our L-Store prototype using row-wise storage layout, which we refer to as L-Store (Row).⁹ In particular, we compared the single-threaded scan performance (for low contention and 4K range size) of L-Store using both row and columnar layouts in the presence of when there is no updates or when there are 16 concurrent update threads (as shown in Table 5). As expected, the scan performance of L-Store (Column) is substantially higher than L-Store (Row) by a factor of 2.75 \times and 4.56 \times , with and without updates, respectively. Also note that we did not enable column compression for L-Store (Column), otherwise even a higher performance gap would be observed because in column stores, an average of 10 \times compression is commonly expected [5, 36].

We further conducted an experiment with only point queries (on a table with 10 columns), where each transaction now consists of 10 read statements, and each read statement may read 10% to 100% of all columns (as shown in Table 6). As expected, the performance of any column store is deteriorated as more columns are fetched. When reading only 10-20% of columns, L-Store (Column) exhibit a comparable throughput as L-Store (Row); however, as we increase the number of fetched columns, the throughput is decreased. But, even in the worst case when all columns are fetched, the throughput only drops by 33%. However, the prevalent observation is that rarely all columns are read or updated in either OLTP or OLAP workloads [1, 5, 36]; thus, given the substantial performance benefit of columnar layout for predominant workloads, then it is justified to expect a slight throughput decrease in rare cases of when point queries are forced to access all columns.

Impacts of comparing lineage-based vs. log-structured storage architecture For completeness, in our prototype, we also implemented log-structured merge-tree (LSM) [25] storage architecture that is predominant in the distributed key-value stores. In particular, we have based our implementation on LevelDB [21]. In this experiment, we studied the single-threaded scan performance (for low contention) of L-Store (i.e., lineage-based storage architecture) and LSM while having 16 concurrent update threads (as shown in Table 7). As expected, due to the multi-layered structured of LSM, the fine-grained read/write access and scan performance

⁹Notably our proposed lineage-based storage architecture is not limited to any particular data layout; in fact, our technique can be employed even for non-relational data such as document or graph data.

of LSM are substantially lower than L-Store. As a result, L-Store outperforms LSM on update throughput and scan performance by a factor of 5 \times and 2.6 \times , respectively.

7 RELATED WORK

In recent years, we have witnessed the development of many in-memory engines optimized for OLTP workloads either as research prototypes such as HyPer [13, 24], ES2 [6], and ExpoDB [11] or for commercial use such as Microsoft Hekaton [9], Oracle In-Memory [15], VoltDB [38], and HANA [14, 27]. Most of these systems are designed to keep the data in row format and in the main memory to increase the OLTP performance. In contrast, to optimize the OLAP workloads, columnar format is preferred. The early examples of these engines are C-Store [36] and MonetDB [5]. Recently, major big data vendors also started integrating columnar storage format into their existing engines. SAP HANA [10] is designed to handle both OLTP and OLAP workloads by supporting in-memory columnar format. IBM DB2 BLU [29] introduces a novel columnar OLAP engine that is memory-optimized and substantially improves the execution of complex analytical workloads by operating directly on compressed data. In what follows, we shift our focus to the recent developments that aim to bring both OLTP and OLAP capabilities into the same platform.

HyPer, a powerful main-memory system, guarantees the ACID properties of OLTP transactions and supports running OLAP queries on consistent snapshot [13]. The design of HyPer leverages a novel OS-processor-controlled lazy copy-on-write mechanism enabling to create a consistent virtual memory snapshot. Unlike L-Store, HyPer resorts to running transactions serially when the workload is not partitionable. Notably, HyPer recently employed multi-version concurrency to close this gap [24]. IBM Wildfire is a variant of DB2 BLU [29] that is integrated into Apache Spark to support fast ingest by adopting the relaxed last-writer-wins semantics and offers an efficient snapshot isolation on recent, but stale, data by relying on periodic shipment and writing of the logs onto a distributed file system [4]. In the same spirit, BatchDB is based on primary-secondary replication design to efficiently isolate OLTP and OLAP workloads by relying on batch migration of recent updates and executing OLAP queries over recent (but possibly stale) snapshots [23]. The unified storage architecture in L-Store eliminates the need for classical log shipment design and does not restrict reads to stale snapshots. Elastic power-aware data-intensive cloud computing platform (epiC) was designed to provide scalable big data services on cloud [6]. epiC is designed to handle both OLTP and OLAP workloads [7]. However, unlike L-Store, the OLTP queries in ES2 are limited to basic get, put, and delete requests (without multi-statements transactional support). Furthermore, in ES2, it is possible that snapshot consistency is violated and the user is notified subsequently [6].

Microsoft SQL Server currently consists of three unique engines: the classical SQL Server engine designed to process disk-based tables in row format, the Apollo engine designed to maintain the data in columnar format that offers significant performance gain for OLAP workloads [19], and the completely redesigned Hekaton in-memory engine designed to excel at OLTP workloads [9, 17]. Noteworthy, Microsoft has also recently announced

moving towards supporting real-time OLTP and OLAP capabilities [17], which further reinforces our position to support real-time analytics. To support OLTP and OLAP among loosely integrated engines, an intricate foreground routine is proposed to enable a continuous data migration from Hekaton (a row-based engine) into Apollo (a columnar engine) [17]. In contrast, in L-Store, we rely on a single unified columnar engine (without the need for maintaining multiple copies of the data) and, more importantly, our consolidation is based on a novel contention-free merge process that is performed asynchronously and completely in the background, and the only foreground task is pointer swaps in the page directory for pointing to the newly created merged pages.

Oracle offers a novel dual-format option to support real-time OLTP and OLAP, where data resides in both columnar and row formats [15]. To avoid maintaining two identical copies of data in both columnar and row format, an effective “layout transparency” abstraction was introduced that maps data into a set of disjoint tiles (driven by the query workload and the age of data), where a tile could be stored in either columnar or row format [3]. The key advantage of the layout-transparent mapping is that the query execution runtime operates on the abstract representation (layout independent) without the need to create two different sets of operators for processing the column- and row-oriented data. In the same spirit, SnappyData proposed a unified runtime engine to combine streaming, transaction, and analytical processing, but from the storage perspective, it maintains recent transactional data in row format while it ages data to a columnar format for analytical processing [30]. SnappyData employed data aging strategies similar to the original version of SAP HANA [35].

Contrary to the aforementioned efforts, in L-Store, we strictly keep only one copy and one representation of data; thus, fundamentally eliminating the need to maintain layout-independent mapping abstraction and storing data in both columnar and row formats. HANA [14, 27] also strives to achieve real-time OLTP and OLAP engine. Most notably, we share the same philosophy governing HANA that aims to develop a generalized solution for unifying OLTP and OLAP as opposed to building specialized engines. But what distinguishes our architecture from HANA is that we propose a unified columnar storage without the need to distinguishing between a main store and a delta store. We further propose a contention-free merge process, whereas in [14], the merge process is forced to drain all active transactions at the beginning and end of the merge process, a contention that results in a noticeable slow down as demonstrated in our evaluation.

8 CONCLUSIONS

We develop Lineage-based Data Store (L-Store) to realize real-time OLTP and OLAP processing within a single unified engine. The key features of L-Store can succinctly be summarized as follows. Recent updates for a range of records are strictly appended and clustered in its corresponding tail pages to eliminate read/write contention, which essentially transforms costly point updates into an amortized, fast analytical-like update query. L-Store achieves (at most) 2-hop access to the latest version of any record through an effective embedded indirection layer. We introduce a novel contention-free and relaxed merging of only stable data in order to lazily and independently bring base pages (almost) up-to-date without blocking on-going and new transactions. Every base page relies on independently tracking the lineage information in order to eliminate all coordination and recovery even when merging different columns of the same record independently. Lastly, a novel contention-free page de-allocation using epoch-based approach is introduced without interfering with ongoing transactions. We demonstrate that L-Store outperforms In-place Update + History by factor of up to 5.37× for short update transactions while achieving slightly improved performance for scans. It also outperforms Delta

+ Blocking Merge by 7.91× for short update transactions and up to 2.37× for long-read analytical queries.

9 ACKNOWLEDGMENTS

We wish to thank C. Mohan, K. Ross, V. Raman, R. Barber, R. Sidle, A. Storm, X. Xue, I. Pandis, Y. Chang, and G. M. Lohman for many insightful discussions and invaluable feedback in the earlier stages of this work.

REFERENCES

- [1] Anastasia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. Weaving Relations for Cache Performance. In *VLDB '01*.
- [2] Ioannis Alagiannis, Stratos Idreos, and Anastasia Ailamaki. H2O: a hands-free adaptive store. In *SIGMOD'14*.
- [3] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. Bridging the Archipelago Between Row-Stores and Column-Stores for Hybrid Workloads. In *SIGMOD'16*.
- [4] Ronald Barber, et al. Evolving Databases for New-Gen Big Data Applications. In *CIDR'17*.
- [5] Peter A. Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR'05*.
- [6] Yu Cao, Chun Chen, Fei Guo, Dawei Jiang, Yuting Lin, Beng Chin Ooi, Hoang Tam Vo, Sai Wu, and Quanqing Xu. 2011. ES2: A Cloud Data Storage System for Supporting Both OLTP and OLAP. In *ICDE'11*.
- [7] Chun Chen, Gang Chen, Dawei Jiang, Beng Chin Ooi, Hoang Tam Vo, Sai Wu, and Quanqing Xu. Providing scalable database services on the cloud. In *WISE'10*.
- [8] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *SoCC'10*.
- [9] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL Server's Memory-optimized OLTP Engine. In *SIGMOD'13*.
- [10] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dies. 2012. The SAP HANA Database – An Architecture Overview. *IEEE Data Eng. Bull.'12*
- [11] Suyash Gupta and Mohammad Sadoghi. EasyCommit: A Non-blocking Two-phase Commit Protocol. In *EDBT'18*.
- [12] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-Store: a high-performance, distributed main memory transaction processing system. *PVLDB'08*.
- [13] Alfons Kemper and Thomas Neumann. HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *ICDE'11*.
- [14] Jens Krueger, Changkyu Kim, Martin Grund, Nadathur Satish, David Schwab, Jatin Chhugani, Hasso Plattner, Pradeep Dubey, and Alexander Zeier. Fast Updates on Read-optimized Databases Using Multi-core CPUs. *PVLDB'11*
- [15] T. Lahiri, et al. Oracle Database In-Memory: A dual format in-memory database. In *ICDE'15*.
- [16] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In *SIGMOD'16*.
- [17] Per-Åke Larson, Adrian Birka, Eric N. Hanson, Weiyun Huang, Michal Nowakiewicz, and Vassilis Papadimos. 2015. Real-time Analytical Processing with SQL Server. *PVLDB'15*.
- [18] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. 2011. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *PVLDB'11*.
- [19] Per-Åke Larson, Cipri Clinciu, Eric N. Hanson, Artem Oks, Susan L. Price, Srikumar Rangarajan, Aleksandras Surma, and Qingqing Zhou. SQL Server Column Store Indexes. In *SIGMOD'11*.
- [20] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. The Bw-Tree: A B-tree for New Hardware Platforms. In *ICDE'13*.
- [21] LevelDB <http://leveldb.org/>
- [22] Jan Lindström, Vilho Raatikka, Jarmo Ruuth, Petri Soini, and Katriina Vakkila. IBM solidDB: In-Memory Database Optimized for Extreme Speed and Availability. *IEEE Data Eng. Bull.'13*.
- [23] Darko Makreshanski, Jana Giceva, Claude Barthels, and Gustavo Alonso. BatchDB: Efficient Isolated Execution of Hybrid OLTP+OLAP Workloads for Interactive Applications. In *SIGMOD'17*.
- [24] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *SIGMOD'15*.
- [25] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Inf.'96*.
- [26] Oracle Total Recall/Flashback Data Archive.
- [27] Hasso Plattner. The Impact of Columnar In-memory Databases on Enterprise Systems: Implications of Eliminating Transaction-maintained Aggregates. *PVLDB'14*
- [28] T. Rabl, S. Gómez-Villamor, M. Sadoghi, V. Muntés-Mulero, H.-A. Jacobsen, and S. Mankovskii. Solving big data challenges for enterprise application performance management. *PVLDB'12*.
- [29] Vijayshankar Raman, et al. DB2 with BLU Acceleration: So Much More Than Just a Column Store. *PVLDB'13*
- [30] Jags Ramnarayan, Sudhir Menon, Sumedh Wale, and Hemant Bhanawat. SnappyData: A Hybrid System for Transactions, Analytics, and Streaming: Demo. In *DEBS'16*.
- [31] Mohammad Sadoghi, Souvik Bhattacharjee, Bishwaranjan Bhattacharjee, and Mustafa Canim. L-Store: A Real-time OLTP and OLAP System. *CoRR'16 abs/1601.04084*.
- [32] Mohammad Sadoghi, Mustafa Canim, Bishwaranjan Bhattacharjee, Fabian Nagel, and Kenneth A. Ross. Reducing Database Locking Contention Through Multi-version Concurrency. *PVLDB'14*.
- [33] Mohammad Sadoghi, Kenneth A. Ross, Mustafa Canim, and Bishwaranjan Bhattacharjee. 2013. Making Updates disk-I/O Friendly Using SSDs. *PVLDB'13*.
- [34] Mohammad Sadoghi, Kenneth A. Ross, Mustafa Canim, and Bishwaranjan Bhattacharjee. Exploiting SSDs in operational multiversion databases. *Vldb'16*.
- [35] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. Efficient Transaction Processing in SAP HANA Database: The End of a Column Store Myth. In *SIGMOD'12*.
- [36] Mike Stonebraker, et al. C-Store: A Column-oriented DBMS. In *Vldb'05*.
- [37] Michael Stonebraker and Ugur Cetintemel. “One Size Fits All”: An Idea Whose Time Has Come and Gone. In *ICDE'05*.
- [38] Michael Stonebraker and Ariel Weisberg. The VoltDB Main Memory DBMS. *IEEE Data Eng. Bull.'13*

A Hybrid Approach for Alarm Verification using Stream Processing, Machine Learning and Text Analytics

Ana Sima
Zurich University of Applied
Sciences
Winterthur, Switzerland
simn@zhaw.ch

Kurt Stockinger
Zurich University of Applied
Sciences
Winterthur, Switzerland
stog@zhaw.ch

Katrin Affolter
Zurich University of Applied
Sciences
Winterthur, Switzerland
affl@zhaw.ch

Martin Braschler
Zurich University of Applied
Sciences
Winterthur, Switzerland
bram@zhaw.ch

Peter Monte
Sitasys AG
Langendorf, Switzerland
peter.monte@sitasys.com

Lukas Kaiser
Sitasys AG
Langendorf, Switzerland
lukas.kaiser@sitasys.com

ABSTRACT

False alarms triggered by security sensors incur high costs for all parties involved. According to police reports, a large majority of alarms are false. Recent advances in machine learning can enable automatically classifying alarms. However, building a scalable alarm verification system is a challenge, since the system needs to: (1) process thousands of alarms in real-time, (2) classify false alarms with high accuracy and (3) perform historic data analysis to enable better insights into the results for human operators. This requires a mix of machine learning, stream and batch processing – technologies which are typically optimized independently. We combine all three into a single, real-world application.

This paper describes the implementation and evaluation of an alarm verification system we developed jointly with Sitasys, the market leader in alarm transmission in central Europe. Our system can process around 30K alarms per second with a verification accuracy of above 90%.

1 INTRODUCTION

False alarms triggered by sensors of alarm systems pose a significant challenge due to the high costs they incur for all involved parties. On the one hand, false alarms waste expensive police, medical and firefighter resources. On the other hand, Alarm Receiving Centers (ARCs) cannot efficiently prioritise important alarms, because they are overwhelmed with false ones. According to police reports, a large majority of alarms prove to be false [34]. This is often attributed to technical errors, installation errors or user errors. As a consequence, owners of alarm systems end up switching their systems off to avoid the risk of paying for intervention forces deployed as a response to a false alarm.

From a technical perspective, false alarm verification is very challenging, since it requires the combination of three traditionally separate fields, namely stream processing, batch processing and machine learning. Depending on the data sources used for verification, both structured data (originating from the physical security sensors) and unstructured (originating from external sources, such as social media or police news feeds, available in free-text format) should be integrated. Recently, stream and batch processing have been integrated into combined systems such as

Flink [7] or Apache Structured Streaming [44]. However, adding machine learning, unstructured data and a real use-case to the equation makes the problem much harder. Machine learning has proven successful in a wide range of classification and anomaly detection tasks [26]. In particular, a classification model can be trained in order to compute the likelihood that a new alarm is either true or false, based on the history of alarms previously received in the system. Such algorithms have the potential to significantly reduce costs involved by false alarms, by enabling ARCs to focus on the alarms that are most likely true, while reducing the priority and the resources allocated for alarms that are likely false.

In this paper we present our experience in building an end-to-end alarm verification system that combines stream processing, batch processing and machine learning on both structured and unstructured data in an industrial setting. We use state-of-the-art Big Data technology such as Apache Kafka [21], MongoDB [33] and Apache Spark [38]. We show that our models can classify false alarms with over 90% accuracy and can scale up to 30K alarms per second including historical analysis using real alarm data from our industrial partner. Furthermore, we show that our models can be adapted with minimal effort and achieve good performance for similar use cases. For example, we use the same algorithms to train a new model from the history of fire incidents recorded by the cities of London and San Francisco.

The main contributions of this paper are:

- We present an end-to-end application that combines stream processing, batch processing and machine learning in order to uncover false alarms in an industrial setting. Using a dataset of 350K real alarms from alarm sensors deployed in production, we evaluate 4 machine learning algorithms and show that the best 2 algorithms (random forest and deep neural networks) can classify alarms with over 90% accuracy. This is, to our knowledge, the first study to show the applicability of machine learning techniques for false alarm reduction in the field of physical security, using real data collected from alarm sensors used in production.
- We show that a simple set of generic alarm features (location, time, property type) can be used for similar use cases. By reusing the exact same algorithms implemented for our industrial use case, we yield a verification accuracy of above 80% for the additional datasets from the cities of London and San Francisco.

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

- We discuss how to extend the approach to include a-priori risk factors extracted from external sources, such as news articles and social media postings, that potentially cover incidents related to alarms ("hybrid approach"). These sources usually provide unstructured, free-text data. In our prototype implementation, we focus on reports about fire and intrusion incidents. Even though we had only limited external data available, we increased the accuracy of our classifications from a baseline of 86.56% for the subset of fire and intrusion alarms to 87.56% when including the a-priori risk in the machine learning model.

The rest of this paper is organized as follows: We introduce Related Work in Section 2. In Section 3 we present the industrial use case for alarm management. We discuss the architecture as well as the design of our system in Section 4. We describe our experiments with various machine learning algorithms as well as end-to-end performance results based on stream processing, batch processing and machine learning in Section 5. Finally, we present an extensive list of lessons learned in Section 6 and conclude in Section 7.

2 RELATED WORK

2.1 Stream Processing

Stream and continuous event processing for real-time analytics has been a major topic of the database community for more than a decade with Aurora [9] being one of the pioneers. Other popular systems are Gigascope [13], Esper [16] or Stream Base [40]. Common to these systems is that they provide a declarative query language based on SQL to process data streams. The advantage of these systems is that end users can formulate analytics queries using the expressiveness of SQL rather than learning new APIs. Moreover, since SQL is declarative, the end-users need not care of how they would optimize the system performance since the stream systems can apply query optimization techniques by understanding the query patterns.

Common to all these systems is that they are highly specialized for one particular functionality, namely stream processing with short time windows. However, they are inadequate for combined stream and batch processing since they only focus on stream processing.

2.2 Real-time Data Warehousing

Typical data warehouses of large enterprises are used for reporting, analytical and predictive purposes. In order to optimize query performance, these systems organize data in a star schema [10]. Moreover, data is usually ingested on a daily or subdaily basis. Common to all traditional data warehouses is that they are very efficient for processing historical data but not particularly well suited for processing streams of data.

In order to overcome these problems, recently so-called data stream warehouses have been proposed to handle both big and fast amounts of data within one single system. In other words, the idea is to use one, combined system for stream and historical data analytics. Examples of such systems include Borealis [1], DataDepot [17], DejaVu [14], Moira [6] or TruViso [24].

The advantage of these systems over streaming-only systems is that they can handle combined workloads of both stream and batch processing.

2.3 Combined Stream and Batch Processing

As part of our previous work we have used bitmap indexes to enable stream and batch processing in TelegraphCQ [35]. We have demonstrated the approach for analyzing a large set of network traffic data.

To tackle the problem of combined stream and historical data analytics for more recent Big Data systems, the Lambda architecture [31] was introduced that currently sets the standard in system design for building big data real-time analytics environments. It is trying to provide a solution to compensate latency and waiting time when accessing and analyzing batch processed data through the availability of real-time data streams. However, criticism on the Lambda architecture revolves around the operational complexity of systems implementing this architecture. This does not only include operations of the systems but especially also implementing and maintaining an efficient code base for the two different data processing approaches - stream and batch processing - used in systems built according to the Lambda architecture.

Real-time processing for NoSQL systems has recently been introduced in Muppet [25], SCALLA [29] and Spark Streaming [44]. In particular, structured streaming seems very promising. Another system that provides both stream and batch processing is Apache Flink [7]. However, it is still difficult to smoothly integrate different technologies to develop a system for complex scenarios that can leverage existing legacy systems. A more recent reflection on main memory vs. stream systems can be found in [23].

2.4 Machine Learning for Anomaly Detection

Machine learning has been widely used for classification and anomaly detection. The research closest to ours has been done in intrusion detection systems in the field of computer and network security [26]. Subsequently, in order to make these systems usable in practice, a lot of work has focused around means to reduce their false positives [27], [4], [12]. The recent shift of the alarm industry towards IoT and smart connected sensors has opened the path for applying the same algorithms in a relatively new context, namely that of physical security, which is our focus in this paper. There is to-date surprisingly little published data on the effectiveness of these techniques for physical IP-connected alarm systems.

Most of the related work published either in research papers or in industrial patents aims at reducing false alarms by means of verification through a secondary channel - e.g. a video camera or additional sensors, such as temperature, shock or vibration sensors. In [30], an intelligent home security system based on the ZigBee protocol is presented. The system detects false alarms by means of image processing from surveillance cameras. However, we do not rely on any other information apart from alarm device properties, the type of supervised premise, location and time.

Similarly, a patent issued by Honeywell AG presents a system that reduces false alarms in a home security system by using information provided by additional sensors, such as an acoustic glass break sensor, shock sensor and vibration sensor [3]. More recently, Honeywell extended their systems with a video-verification step to reduce the number of false alarms [19]. In contrast, our approach has the advantage of being more generic, given that it relies only on information provided by basic sensors and a model trained offline (from the history of alarms received

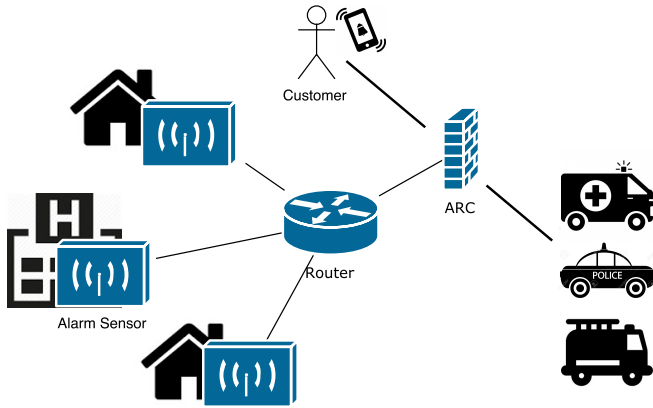


Figure 1: Alarm System Architecture. ARC = Alarm Receiving Center.

from these same sensors), even in the lack of more contextual information (camera images, weather sensors etc).

An interesting approach is presented in [32], where the most suitable machine learning algorithm is chosen adaptively based on the performance of the currently used one - this could be an interesting path for future work in our system, as we have already implemented 4 machine learning pipelines, therefore we would only require the logic to adaptively choose among these at run-time. Another angle to consider would be a majority vote among the different classifiers, providing the overall verification and probability as an aggregate of the information provided by all 4 classifiers.

In a recent publication, Spark Streaming [44] and Apache Kafka [21] were used to detect anomalies in Big Data streams by applying various metrics based on entropy theory and Pearson correlation [36]. In our project we partially build on these results. Our initial machine learning experiments showed promising results [37].

3 ALARM VERIFICATION USE CASE

In a typical security installation, transmitting an alarm originating from a sensor (e.g. motion or smoke detector) to a security organization involves a chain of equipment and people. A simplified view of this setup is shown in Figure 1. An alarm triggered at a supervised premise (a home or enterprise) will reach the security organization (also called Alarm Receiving Center - ARC), where the alarm is handled by one of multiple operators who take predefined actions based on the so called action plan which was previously elaborated together with the customer. This usually involves trying to contact the customer by phone to verify the alarm. This is an important step because more than 90% of all alarms are false positives [34]. If the operator is not able to reach out to the customer or the alarm was verified, he sends out intervention personnel (police, ambulance or firefighters) to go on-site.

The high amount of false alarms makes alarm handling costly. Certas AG, one of the major companies in the alarm monitoring market, processes nearly 5 million alarms and over 2 million phone calls a year, as they report in the Alarm Management Symposium in 2017 [11]. Our industrial partner, Sitasys, operates a platform that connects hundreds of such monitoring centers. Today, a large number of messages are generated from a relatively small amount of sources (like fire sensors or motion detectors).

With the advent of Smart Homes and IoT technologies, the number of sensors and therewith the number of alarms is expected to increase drastically. Meanwhile the demand for security also increases. The security industry in Austria, for example, has grown almost 45% between 2010 and 2016 (as indicated in the yearly security book 2017 published by VSO [42]). With these trends, the monitoring centers risk to get flooded with alarm messages. Keeping in mind that the rate of false alarms is above 90%, it becomes clear that there is a need for improvement.

Uncovering false alarms through machine learning is challenging, since there may not even be a clear definition on whether an alarm is worthy of investigation or not, thus rendering a 100% accuracy a hypothetical goal. However, by changing the process of alarm handling, there might be a way to use predictive modelling in a safe way in order to reduce costs significantly. The way this could be achieved is to transfer the verification of the alarm partly to the customer. The idea is to transmit alarms with a high probability of being false positives to the customer's mobile phone first. The customer can then decide within a time window whether the alarm is real or false and whether it should be sent to the alarm receiving center. Only alarms with a high probability of being true (and those for which a timely answer could not be received directly from the customer), are forwarded to the monitoring center, which can then send out intervention personnel.

With this approach, the number of alarms arriving at the monitoring center decreases, while the handling of the particular alarm becomes more effective, since the manual verification can be omitted. With this self-monitoring solution, the customer can actively take part in the alarm processing chain, which will decrease the workload at the monitoring center and consequently potential errors caused by overwhelmed operators.

Furthermore, the probability for true and false alarms can be used by the monitoring center in order to effectively prioritize alarms. This is especially helpful in the case of large events, which generate a spike of messages that need to be processed fast. An effective prioritization of alarms allows a more effective use of intervention personnel. This ultimately benefits the customer as well, because it reduces the fees he has to pay if the police or fire brigades respond to false alarms repeatedly.

The solution envisioned by Sitasys involves an online portal called "My Security Center". Using "My Security Center" the customer can configure the threshold for the probability of alarms being classified as true. The customer thus decides which alarms should be sent to the monitoring center and which should be sent to his mobile phone first. He can also decide not to send technical alarms, like connection interruptions, to the monitoring station at all. Based on his settings, the alarm handling can be offered for about 40% of the price that is currently common in the market - without any sacrifice concerning security. Since "My Security Center" allows dynamic changes of the rules how and where alarms are being transmitted, it will also allow the offering of more custom tailored services.

4 SYSTEM DESIGN

In this section we motivate the main workflow, the system architecture for our alarm verification application and discuss our design choices.

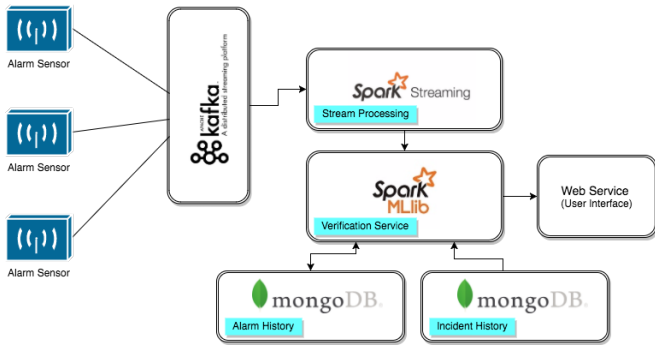


Figure 2: System design consisting of four components: (1) Stream Processing, (2) Batch Processing (Alarm History), (3) Machine Learning (Verification Service) and (4) Hybrid Approach (Incident History).

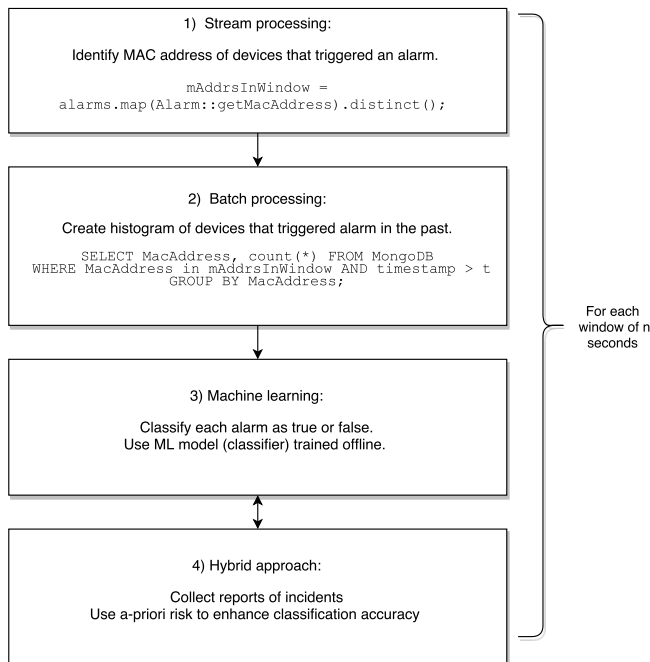


Figure 3: Workflow diagram.

4.1 Workflow

The workflow for alarm verification, shown in Figure 3, consisting of stream processing, batch processing and machine learning, can be characterized as follows.

The stream processing part identifies all devices that trigger an alarm within a certain observation period (the streaming window). As part of the batch processing part, all devices that triggered an alarm are analyzed in more detail by producing a histogram of the number of alarms starting from a specific time t . Finally, for all the new alarms in the given time window, a machine learning algorithm verifies whether the alarm is true or false, based on a classifier trained periodically offline (for example, once per day during idle periods, such as after midnight).

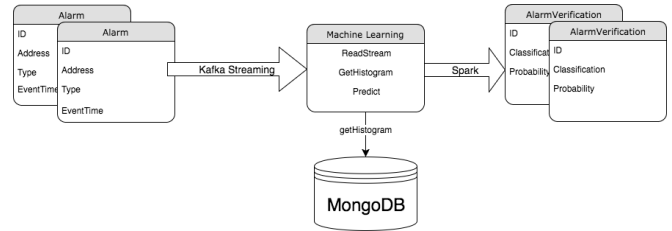


Figure 4: System Architecture and Process Flow

4.2 System Architecture

An overview of the system we developed for handling the above mentioned workflow is shown in Figure 2. The four main components we developed are:

- (1) Streaming Component.** This component is responsible for transmitting and receiving alarms. The simplified format of an alarm sent by a Sitasys sensor through a Kafka stream is shown on the left hand-side of Figure 4. We chose Apache Kafka [21] for this component as it is the state-of-the-art distributed streaming platform, highly scalable and also easy to integrate with Spark. We coupled Spark with Kafka through Direct Dstreams [44], which offers exactly-once semantics "out-of-the-box". This is crucial in our case in order to ensure that we neither miss an alarm, nor process the same one multiple times. For more details refer to [22].
- (2) Batch Component (Alarm History).** This component is responsible for long-term storage of alarms and for doing batch analytics on the history of alarms. For this component we chose to use MongoDB [33], both because of its flexibility (we can store alarms directly as JSON-like documents and query by fields, such as by location of the alarm in order to compute a histogram) and because of its scalability.
- (3) Machine Learning Component (Verification Service).** The reception of a new alarm through the stream immediately triggers the computation of a classification (true/false alarm) and the associated probability (confidence), based on a machine learning model trained offline. The verifications will be used by the Alarm Receiving Centers in order to prioritize incidents where an intervention (police or fire department) is highly likely to be required. We chose to implement this component using Apache Spark, first because it is easy to scale-out when required - for example, if more customers install alarm systems - and second, because of its fault tolerance guarantees. Coupling Kafka with Spark results in an exactly-once semantics streaming application.
- (4) Hybrid Approach (Incident History).** For the hybrid approach we collect reports about fire and intrusion incidents in Switzerland. The incidents are reported as textual data, for example in RSS feeds, Twitter messages or web-pages (see Figure 5). The goal is to use this historical data in order to calculate an a-priori risk factor per each location (village or city in Switzerland) and incorporate it in the machine learning model. Our pipeline collects as many reports as possible and then filters those pertaining to relevant topics (fire and intrusion), based on a set of keywords defined in the pipeline. Each incident report is then annotated with a time and location, extracted directly

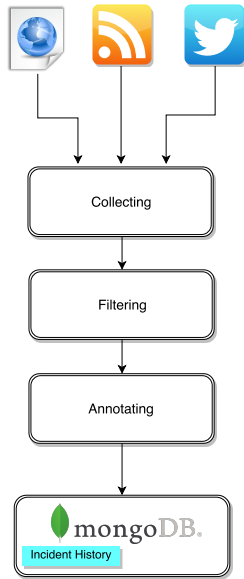


Figure 5: Schema of the incidents history pipeline. In a first step, reports from different sources, such as Twitter, RSS feeds and web pages are collected. Next, reports related to relevant topics (e.g. fire, intrusion) are selected (filtered). The remaining relevant reports are annotated with a date, a location and a language and saved in MongoDB.

from the textual data or from the metadata (if available). Finally, similar to the alarm history, we store the incident history in MongoDB. The a-priori risk factors are defined as the number of incidents per capita on location level.

4.3 Reflection on Design Choices

One of the major design choices was the architecture and technology used for streaming processing, batch processing and machine learning. For stream processing the main options available were Apache Storm [39], Apache Kafka and Apache Spark Streaming [44]. We have decided for the combination of Apache Kafka and Spark Streaming due to the good integration and the scalability of both systems. Even though Storm allows topologic modelling of streaming tasks, we decided against it since our application does not have a complex dependency between tasks.

In order to combine stream and batch processing, the design choices would be either the more traditional Hadoop stack for batch processing combined with Storm for streaming processing, or the more recent, in-memory, Apache Spark technology. We have chosen the latter due to its tighter integration of functionality (stream and batch processing as well as machine learning available in a single framework). This was an important advantage for our industry partner Sitasys, in order to decrease the complexity of the overall system architecture, as well as to reduce maintenance costs and required skills of their workforce.

At the start of our project, Spark Structured Streaming was marked experimental, therefore not yet production ready, hence we decided against it. Moreover, our industry partner has a large collection of alarm data stored in MongoDB which should be leveraged. After analyzing the integration of Spark with MongoDB, we decided for this design option since it allowed us to re-use technology already existing at our industrial partner and

to effectively combine it with state-of-the art stream processing. Moreover, MongoDB is flexible to schema changes, which makes it a better option for long-term storage than a traditional Relational Database, given that in our use case the structure of an alarm differs across sensor types and even across software updates. Using MongoDB allowed our industry partner to easily ingest data from new alarm installations, even when the structure of the new alarm data did not match the structure from previous installations. In our experiments we achieved satisfying scalability results of MongoDB queries for large datasets. For more details see Section 5.

Using Spark ML for machine learning was a natural choice since it is readily integrated within the Spark technology stack. However, since Spark ML did not provide deep learning algorithms at the start of our project, we used various other deep learning frameworks such as DeepLearning4J [15] as well as Theano [41] and Lasagne [28].

5 EVALUATION

In this section we first describe the alarm datasets we used for our experiments, namely from Sitasys, as well as from the cities of London and San Francisco. Next, we describe the incidents dataset we used for the hybrid approach to enrich the Sitasys alarm dataset. This is followed by a description of our machine learning experiments in order to classify false alarms and a description of our experiments using the hybrid approach in order to improve the machine learning pipeline. Finally, we present the end-to-end evaluation of our system, which includes stream processing, batch progressing, as well as machine learning. Our results show that using production data we can process 30K alarms per second with an accuracy of above 90%.

5.1 Alarm Datasets

In order to build and evaluate a model for false alarm verification, we started by analyzing the alarm data provided by our industrial partner, Sitasys. This data is presented in Section 5.1.1. First, we selected the set of features best suited for verification. We describe this in Section 5.3. Then, in order to evaluate how well we can extrapolate using this set of features for similar use cases, but also to see how well our algorithms scale, we looked for larger datasets available online. We identified two candidates, namely the London Fire Brigade Data and the San Francisco Fire Department Data. The most relevant features from all 3 datasets are shown in Table 1. In the next sections we describe each dataset in more detail.

5.1.1 Sitasys Production Data. Real alarm data from October 2015 to April 2016 was collected and anonymized by our industrial partner Sitasys, gathering a total of 350K alarms in roughly equal proportions of true and false alarms. The main types of information provided are location (ZIP code), device address (MAC and IP address), timestamp, alarm duration, type of incident (fire, intrusion, etc.) and a few other sensor-specific information (type of sensor, software version, etc). The ObjectType feature classifies the type of supervised premise the alarm originates from: industrial, residential etc. The location information was anonymized (hashed) for privacy reasons. One important challenge we faced when using this data is the lack of real labels (i.e. indications about true and false alarms). These could not be provided to us in due time from the Alarm Receiving Centers that register them. However, in collaboration with Sitasys we have defined a heuristic to infer the labels, which is to consider the duration of

Dataset	Location	Time	Type of Location	Incident Type	Label
Sitasys	ZIP code	Timestamp	ObjectType	Alarm Type	Alarm Duration
London	ZIP code	Date/TimeOfCall	PropertyType	PropertyCategory	Incident Group
San Francisco	Zip code Of Incident	ReceivedDtTm	-	Call Type	Call Final Disposition

Table 1: Features of the three data sets

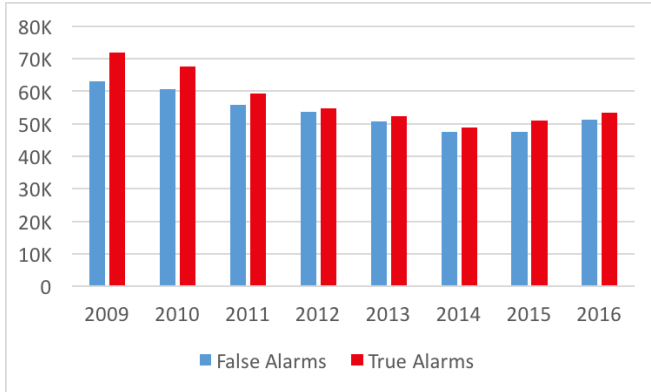


Figure 6: London Fire Brigade Statistics

the alarm as a threshold - the more quickly the alarm was reset after being triggered, the higher the likelihood that the alarm was false, given that the customer was able to reset it in a short period of time. We chose several different values for the alarm reset duration threshold between 1 and 10 minutes. We used 50% of the alarms for training (roughly equal amount of true and false alarms) and the remaining 50% for testing. Moreover, we were interested in the response time for an incoming alarm on a data stream, therefore we also simulated a stream of new alarms from the testing data and tested the performance of our verification system. Details are given in Sections 5.3 and 5.5.

5.1.2 London Fire Brigade Data. The London Fire Brigade (LFB) is the busiest fire and rescue service in the United Kingdom and one of the largest firefighting and rescue organizations in the world. We used the open data of every incident reported since January 2009, available online¹. The dataset provides information about the location, time and type of incident for all records.

Figure 6 shows the high-level statistics of incident groups between 2009 and 2016, as well as the ratio of false vs. true alarms recorded. In total, 885K incidents were recorded, out of which 430K (48%) were false. This dataset is therefore very convenient to use for our classification algorithms, because the true and false classes are almost equally balanced, which makes it suitable even for algorithms that are very sensitive to unbalanced data (e.g. Random Forest).

This dataset is useful for two purposes: first, it allows us to test hypotheses on a coarser time-scale, since the incidents are recorded from 2009 until today, meaning that for example we can try to draw statistics and make verifications based on the days of the year with peaks of incidents. Second, it serves as a scalability test as the number of incidents is twice as large as those provided by our industrial partner.

5.1.3 San Francisco Fire Department Open Data. In an attempt to extend our study, we also considered the San Francisco Fire Department Dataset (available online²), which contains 4.3 million incidents from the city of San Francisco from the year 2000 until today. We found that, in contrast to the London Fire Brigade Data, the quality of the San Francisco dataset is lower, given that more than half of the records (2.5 million) are not properly labeled, the Call Final Disposition - which denotes the final classification of the incident - marked "other". Only 105K are explicitly labeled as "No Merit" (false alarm), i.e. less than our production data from Sitasys from 2015 and 2016 only. Moreover, as shown in Table 1, there is no entry in the dataset that indicates the type of property, which in our study of the Sitasys alarms proved to be an important feature for the classifier. An added problem is the diversity of the types of incidents recorded. For example, more than half of the entries are medical incidents, which are not present at all in the other two datasets. Around 1 million incidents are alarms and fire incidents and only a small fraction of these are properly labeled.

All in all, in our study we could only consider incidents of type "alarm" and "fire" that have a proper label indicating either true or false alarm. Unfortunately, this only results in around 12K incidents, much less than we initially expected. We report results from this small subset in the next section. Finally, we note that we have tried our classification algorithms against *all* properly labeled incidents (including medical, hazards etc) but for this purpose we did not obtain meaningful results - only around 53% accuracy.

5.2 Incidents Dataset

This dataset is a collection of reports about real fire and intrusion incidents, gathered from online resources, such as RSS feeds, Twitter or relevant web pages (police, fire brigades etc.). This external data is used in order to enrich the knowledge base provided by the *Sitasys Production Data*. In particular, we annotate alarms with an a-priori risk factor, based on their location. Since the *Sitasys Production Data* only consists of data from Switzerland, we focus on collecting reports about Swiss incidents. For example, we collected messages related to incidents from 50 different Twitter accounts (cantonal police, fire brigade departments and others) from January 2015 until end of October 2017. Our pipeline checks, for each message, whether it contains information about intrusions or fire (see Figure 5). Next, it identifies the language, the date and the location of the incident, either from the metadata (if available) or directly from the textual data (the message). However, since the metadata does not contain information about ZIP codes, the granularity of each location is either a city or a village. In turn, the granularity of our alarm data set is slightly more detailed, namely, at the level of ZIP codes. For example, some larger cities, such as Basel and Zurich, have multiple ZIP codes for different districts of the city. Since the incidents dataset does not have the same level of granularity as the alarms dataset,

¹<https://data.london.gov.uk/dataset/london-fire-brigade-incident-records>

²<https://data.sfgov.org/Public-Safety/Fire-Department-Calls-for-Service/nuek-vuh3/data>

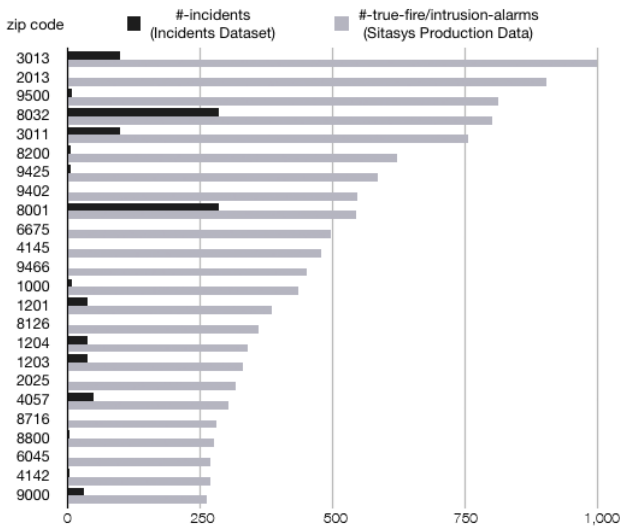


Figure 7: Discrepancy between the amount of incidents and amount of true fire and intrusion alarms in the two data sets.

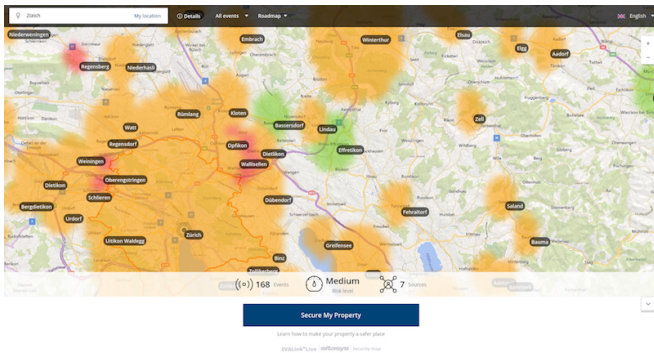


Figure 8: Screenshot of the security map with focus on Zurich (Switzerland). Red areas imply a higher risk level.

we can only approximate an aggregated risk over all districts of a large city with several ZIP codes (see Table 2).

The dataset contains 5,056 descriptions of incidents, out of which 2,743 are in German, 1,516 in French and 797 in English. The corresponding incidents are located in 1,027 distinct cities and villages of Switzerland, covering around 1/4 of all cities and villages in Switzerland. The discrepancy between the number of incidents in the dataset and true fire and intrusion alarms in the *Sitasys Production Data* is shown in Figure 7. For example, the first entry in Figure 6 shows the number of true fire and intrusion alarms for the location with ZIP code 3013 provided from the *Sitasys Production Data* (lower bar shown in light gray). However, the number of reports about fire and intrusion incidents is significantly smaller (upper bar shown in black).

Finally, we use the incidents dataset to build and display a security map of Switzerland, shown here in Figure 8. The figure shows the risk of different areas in the canton of Zurich, Switzerland. Green areas indicate safe regions, whereas yellow indicate medium-risk and finally, red implies higher risk. For a detailed discussion on the calculations of the risk levels we refer to Section 5.4.

ZIP codes Basel	#-true-alarms		#-incidents	
	intrusion	fire	intrusion	fire
4001	43	3	[unknown]	
4051	142	3	[unknown]	
4057	304	0	[unknown]	
4058	0	55	[unknown]	
Total for city of Basel	489	61	10	464

Table 2: Divergence between the number of true alarms in different districts in Basel (Switzerland) in the Sitasys Production Data and the number of incidents collected in the Incident Data. The Incidents Data only contains location information at a coarser granularity (per city / village) than the Sitasys alarm data (per ZIP code).

5.3 Machine Learning

We chose 4 of the most commonly used algorithms for machine learning: Random Forest, Support Vector Machine, Logistic Regression and Deep Neural Networks (DNN). For the first 3 we used the readily available implementations from Spark ML, whereas for DNNs we developed an application using DeepLearning4J [15] as well as Theano [41] and Lasagne [28].

For feature selection we first evaluated which of the alarm fields best separate true from false alarms. We used the Pearson correlation inspired by [36] to find dependencies between features and labels as well as dependencies among features. In addition, we ran a grid search for each algorithm, varying the features used to train the models, and finally selected the following most promising features: location (for privacy reasons we only received hashed location information), day of week, hour of day, alarm type and property type.

Although in our experimental evaluation we only take into account accuracy in terms of number of correct verifications, we note here that, given that our main use case is a decision support system for human operators in the Alarm Receiving Centers, not only is the verification important, but also the probability (confidence) associated with it, as the human operator will likely take a decision according to this metric rather than just the verification.

For our experiments we used the following hardware setup:

- For initial experiments using a single Producer, single Consumer workflow, described in Section 5.5, we used two Intel Xeon E5-2620 machines at 2.4 GHz with 8 GB RAM.
- For multi-node Spark experiments we used a cluster of 4 Intel Xeon E5-2640 CPUs at 2.5 GHz with 16 GB of RAM each.
- For the DNN experiments we used 1 Intel Xeon E5-2650 with 1 Titan X GPU.

5.3.1 Accuracy. As our main use case is to verify false alarms based on the alarm data from our industrial partner, we focused on extracting the features that best separate true from false alarms in the Sitasys dataset. We then investigated how well the equivalent set of features in the open data sets from London and San Francisco can be used to verify false alarms. We show in the next subsections that our approach can be easily transferred to these 2 similar use cases with good accuracy results.

5.3.2 Parameter Tuning. The first important parameter we investigated for the Sitasys alarm dataset was the threshold for

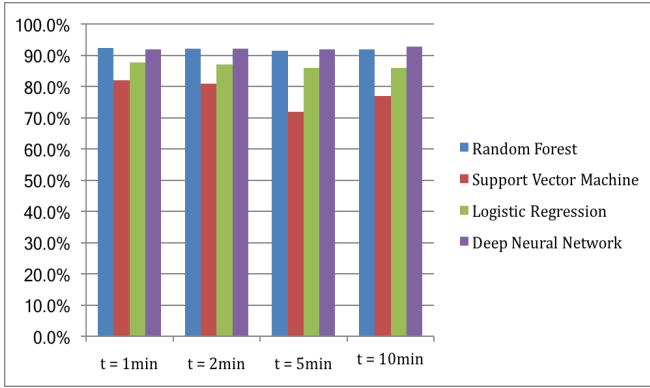


Figure 9: Verification Accuracy vs. Δt (Sitasys Dataset)

the alarm duration. This parameter is used as a heuristic to determine whether the alarm is true or false. For example, when using a threshold of 1 minute, all alarms with a duration smaller than 1 minute are considered false. The intuition is that an alarm that was reset (turned off) within a very short time is likely false (the owner immediately shut it off). In order to evaluate the effectiveness of our machine learning approach, we experimented with various values for delta t ranging between 1 and 10 minutes. The goals of our evaluation were as follows: (1) Evaluate the accuracy of four different machine learning algorithms. (2) Study the impact of various deltas t on the verification accuracy. The idea was to show that the results are stable with respect to changes in the choice of delta t. The results are shown in Figure 9. We can see that on average the best classification quality among all algorithms is achieved with the smallest threshold, of 1 minute, and that moreover in all cases Random Forest and Deep Neural Networks achieve the best performance, of over 90% accuracy, independent of the choice of delta t, which means the accuracy results are stable across changes in the choice of this parameter.

Next, the selection of hyper parameters for each of the learning algorithms (e.g. architecture of neural network) was essential for the verification accuracy. We used grid search to tune the hyper parameters. Tables 3, 4, 5, 6 and 7 show the best ones for each of the 4 algorithms we tested.

Parameter	Value
Maximum depth of a tree	30
Number of trees to train	50

Table 3: Parameters for Random Forest

Parameter	Value
Maximum number of iterations	2,000
Step size	1.0
Mini batch fraction	0.2
Regularization parameter	1e-2
Kernel	Linear
Update Function	Squared L2

Table 4: Parameters for Support Vector Machine

5.3.3 *Training Time.* One important factor we investigated to ensure that our prototype is usable in practice is the training time. Essentially, this determines how fast we are able to

Parameter	Value
Maximum number of iterations	500
Convergence tolerance of iterations	1e-6

Table 5: Parameters for Logistic Regression

Parameter	Value
Maximum number of epochs	10,000
Mini batch size	200
Loss function	Cross Entropy
Update function	Nesterov Momentum
Learning rate	0.1
Momentum	0.9

Table 6: Parameters for Deep Neural Network

Layer	#Nodes	Type	Activation Function
Input	803 Nodes		
Hidden 1	50 Nodes	Fully connected	ReLU
Hidden 2	2 Nodes	Fully connected	ReLU
Output	2 Nodes	Fully connected	Softmax

Table 7: Architecture of Deep Neural Network

rebuild our models, a step that is required periodically (ideally, upon reception of a large enough number of new events, for example once per day). Table 8 shows the training times for our classification algorithms, using the 3 datasets: Sitasys, London Fire Brigade (LFB) and San Francisco Fire Department (SF). The short training time for the San Francisco dataset is explained by the fact that we can only use around 12K incidents properly labeled from the alarm and fire categories. Another observation is that for all the datasets, the smallest training time is required for Logistic Regression, while Deep Neural Networks take much longer to train. Moreover, we use the One Hot Encoding for this algorithm, which means we end up with twice as many input features (around 800) for the Sitasys dataset than for the others (around 300), given that we also use some sensor-specific categorical features in the case of Sitasys (each of the values of these attributes becomes a separate feature when using One Hot Encoding).

Algorithm	Sitays	LFB	SF
Random Forest	600	1200	75
Support Vector Machine	200	480	20
Logistic Regression	100	60	10
Deep Neural Network	5100	2460	60

Table 8: Training Time [sec] for Machine Learning Algorithms

5.3.4 *Accuracy Results.* Figure 10 presents a comparison of the accuracy obtained for the 3 datasets we tested. We can see that the best results are obtained for the Sitasys dataset with a classification accuracy of up to 92% for Random Forest. The promising results can be explained by the fact that, apart from the generic features (Location, Property Type, DayOfWeek and HourOfDay), the Sitasys dataset contains a few other features that can identify technical faults more easily (sensor-specific information), which allows the algorithms to better classify false alarms. By contrast, for the LFB and SF datasets we could only use

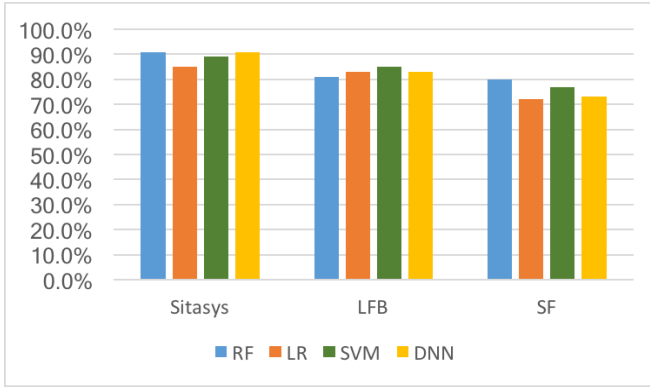


Figure 10: Verification accuracy comparison using four different machine learning algorithms for three Sitasys, London Fire Brigade (LFB) and San Francisco Fire Department (SF) Datasets. RF = Random Forest, LR = Logistic Regression, SVM = Support Vector Machine, DNN = Deep Neural Network

the generic features. However, it is interesting to note that this still results in fairly good accuracy, of around 85%, for the LFB dataset (best result is obtained for the Support Vector Machine) and 80% for the SF dataset (Random Forest). As mentioned previously, the San Francisco dataset does not contain information regarding the type of property an alarm originates from, which could explain the lower accuracy. Furthermore, the volume of the training data we could select from the SF dataset is generally too low (only around 12K alarms).

Another interesting result is that, although there are some differences among the accuracy results for the 4 algorithms we tested, they are never higher than 5%. This is encouraging for two reasons. First, because some algorithms require less training time and less resources to run (Logistic Regression), therefore they could be chosen over the others in case response time is more crucial than accuracy. Second, because more generally this validates our approach, given that the good accuracy is not an artifact of a particular choice of machine learning algorithm, but rather a consequence of the feature selection, which accurately describes the problem we aim to solve. On the other hand, a 5% improvement in classification accuracy (from 85% to 90%) effectively means reducing the error rate by 50%, which means the best algorithms perform significantly better.

5.4 Hybrid Approach

For the hybrid approach we collected descriptions of fire and intrusion incidents from different online resources, such as Twitter, RSS feeds, or web pages collected through a provider of web-based data feeds, webhose.io. Each incident is annotated with the time and location, extracted from the original message or web page. We use this information to calculate an a-priori risk factor for intrusion and fire alarms, based on the number of incidents that happened in a certain location (village or city), normalized by the population size. Next, we evaluate the impact of the inclusion of a-priori risk factors on the accuracy of the machine learning model.

We chose three different ways to include the a-priori risk factors into our machine learning pipeline:

- (1) absolute risk factor
- (2) normalized risk factor

(3) binary risk factor

(1) The absolute risk factor is calculated by dividing the number of incidents found in various resources by the population in the annotated location. (2) The normalized risk factor has a range of 0 to 1, and is calculated as $x' = \frac{x - \min(x)}{\max(x) - \min(x)}$, where x is the absolute risk factor of a location. (3) The binary risk factors are either 0 or 1. The risk factor is 1, if the incident is in the most frequent 25% locations, otherwise the risk factor is 0.

Our efforts for the hybrid approach are still in the early stages, and the data we have collected so far is limited. As a consequence, for the evaluation we only use alarms with a ZIP code where we have corresponding reports about incidents. This reduces the number of alarms from about 350,000 to 130,958 (see Figure 7). As mentioned previously, the granularity of the alarm data is on ZIP code level, while the granularity of external reports about incident data is on city or village level. To analyze the influence of this discrepancy in granularity, we run additional experiments where we only select alarms about small cities or villages that have one ZIP code rather than multiple ones. This reduces the number of alarms from around 130,958 to 37,241 and further the number of incidents from 5,056 to 4,379 (see Table 9). Moreover, the *Sitasys Production Data* provides more alarm types than fire and intrusion. Hence, we needed to filter only those alarms that are triggered due to fire or intrusion.

Table 9 shows the experimental results for alarm classification of four different scenarios (a-d) and three different a-priori risk factors, compared to a baseline approach. The baseline shows the alarm classification accuracy without a-priori risk factors (as reported in Section 5.3). The results are averaged over 10 runs for each experiment. In the scenario (a), using all locations and all alarms, we have around 130,958 alarms. This scenario only shows a small increase of 0.04% of the classification accuracy using the normalized risk factor. Scenario (b) uses all locations, but only the fire and intrusion alarms, which reduces the training data to 24,934 alarms. In this scenario, the results show that the absolute risk factor leads to an increase of 0.22% in accuracy compared to the baseline.

The scenarios (c) and (d) use only locations with a single ZIP code attached. Therefore we make sure that the a-priori risk factor does not contribute wrong information to larger cities with multiple ZIP codes. Out of the total of 130,958 alarms, 37,241 refer to cities with single ZIP codes. This implies that around 2/3 of the alarms are located in larger cities. The results of scenario (c) show an improvement of 0.4% for the absolute risk factor, compared to the baseline. The normalized and binary risk factors also have a slight positive impact. Finally, scenario (d) uses only fire and intrusion alarms for cities with single ZIP codes. Therefore, the number of alarms is reduced to about 10,000 alarms. In this scenario, the impact of applying a-priori risk factors is the strongest, with an increase of 1% compared to the baseline.

Overall, the results obtained by including the external, unstructured data are still preliminary. The scarcity of this data, coupled with an uneven distribution of the reported incidents makes it difficult to measure a significant impact. We still consider the results promising, as they show a) that adding in potentially noisy textual information from third-party sources does not degrade the results even though we are using a limited set of collection and filtering approaches and b) that a small positive impact can already be seen when focusing the analysis on the subset of alarms for which we have matching unstructured data.

scenario	all locations		single ZIP code locations	
	all types (a)	F/I alarms (b)	all types (c)	F/I alarms (d)
<i>baseline</i>	89.35	85.73	87.16	86.56
ARF	89.29	85.95	87.56	87.45
NRF	89.39	85.67	87.41	87.56
BRF	89.31	85.79	87.51	87.48
#-incidents	5,056		4,379	
#-alarms	130,958	24,934	37,241	10,036

Table 9: Accuracy of alarm classification using three different a-priori risk factors and four scenarios: (a) all locations, all alarm types, (b) all locations, only fire & intrusion alarms, (c) single ZIP code locations, all alarm types and (d) single ZIP code locations, only fire & intrusion alarms. ARF = absolute risk factor, NRF = normalized risk factor, BRF = binary risk factor.

5.5 End-to-End Data Processing

Once we have trained and tested the machine learning algorithms, the next step was to build the end-to-end pipeline to integrate machine learning into stream and batch processing. In particular, as soon as an alarm arrives, a machine learning algorithm classifies in real-time whether the alarm is true or false. In addition, historical data analysis is performed on the sensors that triggered the new alarms. The goal of our experiments was to evaluate the maximum throughput of our system, identify potential bottlenecks and to derive lessons learned from building such a production system.

5.5.1 Setup of Streaming System. In order to test the scalability of our prototype, we handcrafted a Producer application, which simulates a stream of new alarms. The stream is created by randomly selecting alarms from the test set (alarms from our production data, that have not been used for training the machine learning model) and writing them into Kafka, at a controlled rate (alarms per second). We aim to evaluate the response time of our system, which runs as a Consumer application.

First, we are interested in measuring the maximum throughput (number of verified alarms per second) on the consumer side. We assume that the training phase has already been completed offline and the model is readily available for computing classifications. Second, we must take into account the maximum latency (system response time) per alarm, as it is critical to ensure that a human operator in the Alarm Receiving Center can get a timely verification result. Currently we set the goal of responding in at most 10 seconds since the reception of the alarm at the ARC.

5.5.2 Identifying Bottlenecks.

Throughput of Producer-Consumer. With a setup as simple as just one producer and one consumer application (running each on a separate machine connected through a 1 GB Ethernet network), we were able to identify several bottlenecks in our system. First, our tests showed that both the producer and the consumer were processing events at an unexpectedly low rate (about 12K alarms produced per second, where one alarm is less than 1KB in size), even if Kafka benchmarks made us expect much higher throughputs. After further investigation we found that the bottleneck in both applications was in fact the serializer used for writing alarms into, and reading them from, the Kafka stream. At first, our implementation used the Jackson serializer, which turned out to be a poor choice for small objects [18], where the Gson serializer is more appropriate. We were surprised to find

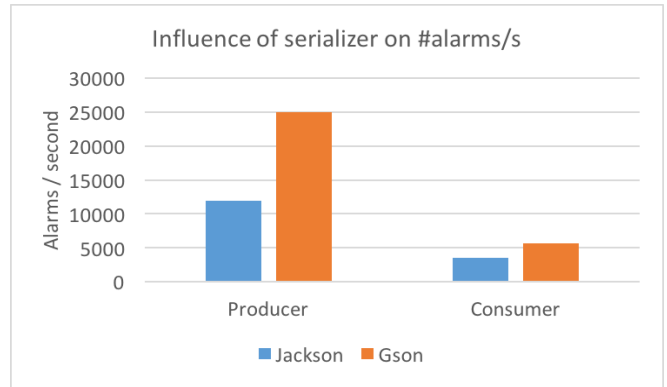


Figure 11: Scalability Jackson vs Gson serializer

that just replacing this led to a 2x speedup in both applications. Figure 11 shows the results. We can see that by switching from the Jackson serializer to the Gson serializer the throughput of the producer more than doubled to about 25K alarms per second. On the consumer side, the increase was slightly less than double. This is due to the fact that the consumer has a higher computing load than the producer.

Detailed Analysis of Consumer. Next we analyzed the computing time of the consumer in more detail (Figure 12). In particular we were interested in the time contribution of stream processing (Spark Streaming), batch processing (MongoDB query) and machine learning (Spark ML). The breakdown of time per component using a 10 second window of alarms shows that the majority of time is spent in the machine learning part (around 80% of the total time) to classify. We also notice that an insignificant factor goes to the historic component (retrieving the histogram of alarms originating from the same addresses as those in the time window). The remaining time goes to the streaming component: first, for deserializing alarms into the native Spark data format, RDDs (Resilient Distributed Datasets [43]), then for extracting all distinct addresses from the RDD etc.

Kafka Optimization. After this step, we further noticed that although our consumer machine has 8 cores, none of the computations were parallelized, although Spark was configured to use all the cores on the local machine. After investigating this we found that by default, Kafka streams are not partitioned, meaning that all RDDs will be processed on a single execution thread. To fix this, there are 2 options available: first, creating multiple streams and reading from them in parallel - this would be useful for the case where, for instance, different customers would be registering alarms to different Kafka streams. However, since for the moment we collect all alarms on the same stream and aim to test per-stream scalability, we chose the second option, which is to configure the repartition number of the Kafka stream, when creating it from the Spark application. In order to test the maximum throughput on the Consumer side, we created multiple threads in the Producer application (to make sure that this does not become the bottleneck) and were in this way able to reach a maximum throughput per consumer of around 30K alarms per second.

6 LESSONS LEARNED

In this paper we have presented an end-to-end system for verifying false alarms in real-time based on a combined stream and batch processing approach. Our results demonstrated that for

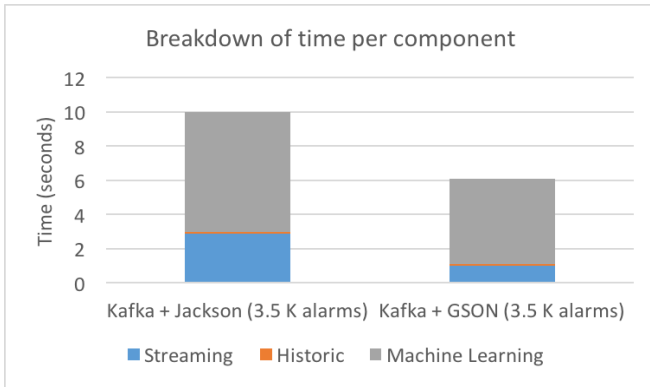


Figure 12: Breakdown of time per component in the Consumer Application

the real production data our machine learning algorithms gained an accuracy of more than 90%. However, even a 99% verification accuracy might not be good enough and could potentially result in missing a fire that burns down a building when no intervention force is deployed. For our industry partner this was the major issue about accepting our approach in a real-world setting. In order to overcome this problem, we introduced the following solutions: (1) The end user (property owner) is in the loop to verify alarms. For instance, before an alarm is sent to the alarm receiving center, the end user can verify it. As an example, let us assume apartment X systematically triggers fire alarms after midnight. A closer inspection shows that the alarms were triggered by the kids since the family has forgotten to de-activate the alarm during the periods the kids went to the bathroom. In this case, the property owner could verify the false alarm and hence no intervention force would be sent. (2) Every alarm is prioritized and evaluated by a human operator. This gives the human operator more time to react to the most critical alarms first and has the potential to drastically decrease human error due to information overload within a short time interval. (3) We provide histograms about historic alarms that help identify recurring problems.

In the next few sections we will report on further lessons learned in the areas of machine learning and Spark processing.

6.1 Machine Learning

- **Provide probability of verification**

Although in our experimental evaluation we only take into account accuracy in terms of number of correct classifications, given that our main use case is a decision support system for human operators in the Alarm Receiving Centers, not only is the verification important, but also the probability (confidence) associated with it. This allows human operators to take a decision according to this metric rather than just the classification. Luckily, most implementations of machine learning (classification) algorithms provide this confidence factor by default. We used the probabilities associated with verifications for the Random Forest or Logistic Regression classifiers from Spark ML as well as for the Neural Networks implementation from the Theano library. Moreover, we provide operators a way to analyze the history of the sensors that triggered alarms in order to get a better understanding about the nature of the new incoming alarms.

- **Keep it simple**

Our experience in testing different machine learning algorithms proved that, while the accuracy was similar for all 4 algorithms we tested, there was a big difference in the training time (from a couple of minutes for Logistic Regression to more than 1.5 hours for Deep Neural Networks). It is therefore crucial to always try out the simplest hypotheses first (even when new advances in the field make it tempting to start with the latest, but much more complex algorithms). This is even more so the case in time-critical applications, where it could be desirable to trade off some accuracy in order to gain in response time.

- **Design for reusability**

While evaluating our prototype we found it extremely useful to have a generic data type that describes our problem, e.g. *LabeledAlarm*, that would not be tied to our particular use case (the data set from Sitasys). We therefore crafted a generic class with categorical features like Location, Property Type, HourOfDay, DayOfWeek, which generally describe alarms (and which can be enriched with other features by extending the class if needed). This minimized the efforts required to adapt and validate the algorithms for new, similar, datasets such as the London Fire Brigade or the San Francisco dataset. Moreover, even if you do not foresee using the algorithms in a new context, it is very likely that the structure of the input data from the initial use case will change over time (in fact, this happened during our project's lifespan), either because of technology changes, software updates or because new components are added in the system (e.g. new types of sensors). Therefore having code that describes the problem in a generic way allows for reusability and adaptability whenever the structure of the input data changes.

6.2 Spark Processing

Although Spark can be very convenient to use, allowing for rapid productivity thanks to its integration of different components in a single platform, it may also lead to suboptimal performance when misconfigured or improperly used. When using Spark we found the following considerations useful:

- **Cache data that will be reused**

Spark's lazy evaluation leads to unnecessary re-computations for data that is not explicitly cached. This side effect is not apparent by just reading the code. We first noticed this problem when evaluating the deserialization mechanism on the consumer side - not only did we notice this step was too slow, it was also executed *twice*, because we reused the input streaming data for both machine learning and for querying historic data, without explicitly caching it.

- **Make use of the monitoring UI**

Spark provides an extremely useful set of statistics, both for batch and streaming, which makes it easy to monitor the application while it is running. The most important statistics we used were the level of parallelism for batch computations and the average delays for stream processing. Both offered insights into points in the application that performed suboptimally.

- **Make sure the parallelism level is the expected one**

One of the problems we noticed by examining the stats in the Spark UI was that input data read from Kafka was always processed serially instead of in parallel. After reading through the documentation, we found that by default,

Kafka streams are not partitioned. Therefore, Spark will not process incoming data in parallel, unless explicitly configured in the code when setting up the Kafka stream.

6.3 Hybrid Approach

The lessons we can draw from our experiments with the hybrid approach are still limited. We believe there is great potential in integrating information from third-party sources into the verification, but to fully leverage this potential, substantial additional research is needed. In this spirit, the following lessons should be read more as suggestions on how to improve on our initial approach:

- **Integrate as many external sources as possible**
Thus making sure the sources cover the alarms as broadly as possible. We have shown that the highest impact can be measured when restricting analysis on the alarms that have corresponding incidents in the external sources. Moreover, our preliminary results show that this has the potential to positively impact classification accuracy.
- **Localize incidents with finer granularity**
This will allow combining incidents and alarms more directly, which should be especially beneficial in heavily populated (urban) areas, where a-priori risks for incidents such as intrusions may vary significantly from one area (neighborhood) to the next.

7 CONCLUSIONS

In this paper we presented the design and evaluation of an alarm verification system using real data from an industry application. The problem is very challenging since it requires a combination of stream processing, batch processing and machine learning. We have built the system using Spark Streaming (stream processing), MongoDB (batch processing) and Spark ML (machine learning). Our experiments with various machine learning algorithms show that the system can classify alarms with an accuracy of more than 90% at a streaming rate of about 30K alarms per second, including historical data analysis. To further extend our system, we also presented preliminary results of an integration of unstructured data to increase the classification accuracy. We concluded with an extensive list of lessons learned that give insights for both academics and practitioners who want to build a similar system.

8 ACKNOWLEDGMENTS

The work was supported by the Swiss Commission for Technology and Innovation (CTI) under grant 18602.1 PFESES. We also want to thank our collaborators Jan Stampfli (formerly at ZHAW), Ilya Kluev (Sitasys), Alexander Gromov (Sitasys), Pascal Hulalka (Sitasys), Markus Prölss (Sitasys) and Jürg Denecke (formerly at Sitasys) for valuable inputs to this paper.

REFERENCES

- [1] D. J. Abadi, M. Balazinska, M. Cherniack et al., The Design of the Borealis Stream Processing Engine, *CIDR*, 2005
- [2] D. J. Abadi, U. Cetintemel, M. Cherniack et al., Aurora: A New Model and Architecture for Data Stream Management, *VLDBJ*, 12(2):120-139, 2003
- [3] R. Adonailo, T. Li, and D. Zakrewski. False alarm reduction in security systems using weather sensor and control panel logic, May 15 2007. US Patent 7,218,217.
- [4] A. Alharby and H. Imai. Ids false alarm reduction using continuous and discontinuous patterns. In *ACNS*, pages 192-205. Springer, 2005.
- [5] Apache Flink project. <http://flink.apache.org/>. Accessed: 2017-07-28.
- [6] M. Balazinska, Y. Kwon, N. Kuchta et al., Moirae. History-Enhanced Monitoring, *CIDR*, 2007
- [7] P. Carbone, S. Ewen, S. Haridi, A. Katsifodimos, V. Markl, and K. Tzoumas. Apache Flink: Stream and Batch Processing in a Single Engine. *Data Engineering*, 2015.
- [8] C.-Y. Chiu, Y.-J. Lee, C.-C. Chang, W.-Y. Luo, and H.-C. Huang. Semi-supervised learning for false alarm reduction. *Advances in Data Mining. Applications and Theoretical Aspects*, pages 595-605, 2010.
- [9] D. Carney, U. Cetintemel and M. Cherniack et al., Monitoring Stream - A New Class of Data Management Applications, *VLDB*, 2002
- [10] S. Chaudhuri, U. Dayal, An Overview of Data Warehousing and OLAP Technology, *SIGMOD Record*, 26(1):65-74, 1997
- [11] Presentation by reto fiechter, certas ag, leiter geschäftstelle zurich at alarm management symposium 2017, <https://save.ch/001/events/alarmanagement-2017-plus/>.
- [12] C.-Y. Chiu, Y.-J. Lee, C.-C. Chang, W.-Y. Luo, and H.-C. Huang. Semi-supervised learning for false alarm reduction. *Advances in Data Mining. Applications and Theoretical Aspects*, pages 595-605, 2010.
- [13] C. D. Cranor, T. Johnson and O. Spatschek et al., Gigascope: A Stream Database for Network Applications, *SIGMOD*, 2003.
- [14] N. Dindar, B. Gifford, P. Lau et al., DejaVu: Declarative Pattern Matching over Live and Archived Streams of Events, *SIGMOD*, 2009
- [15] DeepLearning4j, <https://deeplearning4j.org/>, Accessed: 2017-07-19
- [16] EsperTech Inc., Esper: Event Processing for Java, <http://espertech.com/products/esper.php>, Accessed: 2017-07-19
- [17] L. Golab, T. Johnson, J. S. Seidel, et al., Stream Warehousing with DataDepo, *SIGMOD*, 2009
- [18] J. Dreyfuss. Benchmarking json libraries, available online: <http://blog.takipi.com/the-ultimate-json-library-json-simple-vs-gson-vs-jackson-vs-json/>, may 28, 2015.
- [19] Honeywell, Honeywell Adds Video Alarm Verification To Key Connected Building Offerings <https://www.honeywell.com/newsroom/pressreleases/2017/02/honeywell-adds-video-alarm-verification-to-key-connected-building-offerings>, Accessed: 2017-07-19
- [20] Intel, Streaming SQL for Apache Spark, <https://github.com/Intel-bigdata/spark-streaming-sql>, Accessed: 2017-07-19
- [21] Apache Kafka, A High-Throughput Distributed Messaging System, <http://kafka.apache.org/>, Accessed: 2017-07-19
- [22] Kafka-spark streaming integration guide, available online: <https://spark.apache.org/docs/2.0.0/streaming-kafka-integration.html#approach-2-direct-approach-no-receivers>.
- [23] A. Kipf, V. Pandey, J. Bittacher et al., Analytics on Fast Data: Main-Memory Database Systems versus Modern Streaming Systems *EDBT*, 2017
- [24] S. Krishnamurthy, M. J. Franklin, J. Davis et al., Continuous Analytics over Discontinuous Streams, *SIGMOD*, 2010
- [25] W. Lam, L. Liu, S. Prasad et al., Muppet. MapReduce-style Processing of Fast Data, *PVLDB*, 5(12):1814-1825, 2012
- [26] T. Lane and C. E. Brodley. An application of machine learning to anomaly detection. In *Proceedings of the 20th National Information Systems Security Conference*, volume 377, pages 366-380. Baltimore, USA, 1997.
- [27] K. Law and L. Kwok. Ids false alarm filtering using knn classifier. *Information Security Applications*, pages 114-121, 2005.
- [28] Lasagne, <http://lasagne.readthedocs.io/en/latest/index.html>, Accessed: 2017-07-19
- [29] B. Li, E. Mazur, Y. Diao et al., SCALLA: A Platform for Scalable One-Pass Analytics Using MapReduce, *IACM TODS*, 37(4):27, 2012
- [30] A. Longheu, V. Carchiolo, M. Malgeri, and G. Mangioni. An intelligent and pervasive surveillance system for home security. *International Journal of Computers Communications & Control*, 7(2):312-324, 2014.
- [31] Nathan Marz, James Warren. Big Data: Principles and best practices of scalable realtime data systems. *Manning Publications, 1st edition*, October 2013.
- [32] Y. Meng and L.-f. Kwok. Adaptive false alarm filter using machine learning in intrusion detection. *Practical applications of intelligent systems*, pages 573-584, 2012.
- [33] MongoDB, <https://www.mongodb.com/>, Accessed: 2017-07-19
- [34] Quadragard Einbruchschutz, <https://testseitequadragard.jimdo.com/wissen/einbruchschutz-1/>, Accessed: 2017-07-19
- [35] F. Reiss, K. Stockinger, K. Wu, A. Shoshani, J.M. Hellerstein Enabling real-time querying of live and historical stream data. *SSDBM*, 2007
- [36] L. Rettigy, M. Khayatiy, P. Cudre-Mauroux et al. Online Anomaly Detection over Big Data Streams, *IEEE Big Data*, 2015
- [37] J. Stampfli and K. Stockinger Applied data Science: Using Machine Learning for Alarm Verification: A novel alarm verification service applying various machine learning algorithms can identify false alarms. *ERCIM News*, 107:10, 2016
- [38] Apache Spark, <https://spark.apache.org/>, Accessed: 2017-07-19
- [39] Apache Storm project. <http://storm.apache.org/>. Accessed: 2017-07-19
- [40] StreamBase Inc., StreamBase: Real-Time, Low Latency Data Processing with a Stream Processing Engine, <http://www.streambase.com>, Accessed: 2017-07-19
- [41] Theano, <http://deeplearning.net/software/theano/>, Accessed: 2017-07-19
- [42] VSO Security yearbook, published by the Verband der Sicherheitsunternehmen Österreichs (VSO): https://vsoc.at/files/1_jahrbuch_sicherheit_doppelseitig.pdf.
- [43] M. Zaharia, M. Chowdhury, T. Das et al., Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing, *USENIX*, 2012
- [44] M. Zaharia, T. Das, H. Li, et al., Discretized streams: Fault-tolerant Streaming Computation at Scale. *ACM SOSP*, 2013.

Efficient Secure k -Nearest Neighbours over Encrypted Data

Manish Kesarwani
IBM Research India
Bangalore, India
manishkesarwani@in.ibm.com

Akshar Kaul
IBM Research India
Bangalore, India
akshar.kaul@in.ibm.com

Prasad Naldurg
IBM Research India
Bangalore, India
pnaldurg@in.ibm.com

Sikhar Patranabis
IIT Kharagpur
India
sikhar.patranabis@iitkgp.ernet.in

Gagandeep Singh
IBM Research India
Bangalore, India
gagandeep_singh@in.ibm.com

Sameep Mehta
IBM Research India
Delhi, India
sameepmehta@in.ibm.com

Debdeep Mukhopadhyay
IIT Kharagpur
India
debdeep@cse.iitkgp.ernet.in

ABSTRACT

Enterprise customers of cloud services are wary of outsourcing sensitive user and business data due to inherent security and privacy concerns. In this context, storing and computing directly on encrypted data is an attractive solution, especially against insider attacks. Homomorphic encryption, the keystone enabling technology is unfortunately prohibitively expensive. In this paper, we focus on finding k -Nearest Neighbours (k -NN) directly on encrypted data, a basic data-mining and machine learning algorithm. The goal is to compute the nearest neighbours to a given query, and present exact results to the clients, without the cloud learning anything about the data, query, results, or the access and search patterns. We describe a novel protocol in the two-party cloud setting, using an underlying somewhat homomorphic encryption scheme. In comparison to the state-of-the-art protocol in this setting, we provide asymptotically faster performance, without sacrificing any security guarantees. We implemented our protocol to demonstrate that it is efficient and practical on large and relevant real-world datasets and study how it scales well across different parameters on simulated data.

1 INTRODUCTION

Finding k -Nearest Neighbours (k -NN) is viewed as one of the simplest data mining algorithms for discovering patterns. It is non-parametric, making no assumptions about underlying data distributions. It is also a lazy learning technique, where no attempt is made to generalize the data until a query is presented. The goal of a k -NN algorithm for a given query is to find its k “nearest” neighbours according to a suitable measure of nearness or distance. It has applications in finding candidate patterns for image segmentation, location-based search, and in the classification of symptoms and diagnosis over medical records, to name a few. These patterns may be subsequently used as inputs for more sophisticated learning algorithms.

In the context of cloud computing services, client data records stored on clouds are increasingly confidential in nature, from customer transactions, order histories, credit card numbers and other personally identifiable information (PII). Algorithms, and

especially algorithmic parameters, e.g., in recommendation systems, are also proprietary and sensitive. Inadvertent or unauthorised disclosure of data or computation can have serious legal or business consequences. In order to protect the confidentiality of sensitive information, clients can store encrypted data in data centres. However, it is not cost effective to import back all the data to the client, decrypt and perform computations, as this negates the advantages of moving to the cloud platform in the first place.

The Secure k -NN problem for encrypted data has been a topic of active research [13, 14, 34, 39–41]. The goal is to compute k -NN over encrypted data stored in the cloud, given a query, with the requirement that the cloud provider does not get any information about the plaintext values in the database, the query, the results, the access patterns during query evaluation, and search patterns. The technology to work effectively on encrypted data is provided by a class of encryption schemes called homomorphic encryption (HE), which allow one to compute arbitrary functions on encrypted data and produce encrypted results. Clients, who have the secret keys can decrypt and use these results safely, without revealing the data or results to the servers. Fully HE [16] schemes are expensive and impractical for now [12]. Existing solutions for Secure k -NN, therefore, work in one of two models: the centralised (private) or the two-party (public with multiple servers) cloud model. These solutions rely on a combination of partially or somewhat homomorphic schemes (PHE or (S)HE) for computation [14, 28, 39] of simpler mathematical functions, such as order preserving encryption, homomorphic addition, computation of limited degree polynomials, etc., which are faster to compute. These choices impose different tradeoffs on computation and communication overheads, and provide different security guarantees, making this a rich field for new research.

We present a new protocol in the two-party honest-but-curious cloud model for computing k -NN on encrypted data, which is *asymptotically faster* than the current state-of-the-art protocol [14], without compromising on strong security guarantees. We use LFHE (leveled fully homomorphic encryption) [9], and compute squared Euclidean distances directly on encrypted data. In order to compute the ranking among the distances, we transform the distances suitably to preserve their order and offload the comparison to a federated public cloud, who has secret keys. Since this cloud has access only to transformed results of computations performed on plaintext data, we show that in spite

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

of this knowledge, this honest-but-curious cloud does not learn anything useful about the original database, the results, or the query.

We implement our protocols and run experiments on two real-world datasets from the UCI machine learning repository¹: the cervical cancer (risk factors) dataset, and the default of credit card clients dataset. Both datasets have a large number of dimensions (32 and 23 respectively). Our implementation shows that we are able to find k -NN very efficiently: 166 s on the cancer data and 373 s for the credit card dataset for our secure version for an 8-NN query, making our implementation practical for real-world applications. This trend is also echoed in the simulation results on synthetic datasets, to study the sensitivity of our protocol to various parameter choices empirically.

The rest of the paper is organized as follows: In Section 2 we present background information about our adversary and cloud models as well as a brief tutorial on homomorphic encryption. In Section 3 we present our protocol in detail, followed by a security analysis in Section 4. We describe our implementation and experiments in Section 5, relevant related work in Section 6, and conclude with future work in Section 7.

2 BACKGROUND

In this section, we present our two-party cloud architecture and describe our adversary model and trust assumptions. We also specify the Secure k -NN problem in terms of what functions need to be computed on encrypted data, and briefly introduce homomorphic encryption, with the goal of making this paper self-contained.

2.1 Cloud Architecture

The security implications of outsourcing data or computation to cloud servers need to be studied carefully. There are many deployment choices, including private clouds, public clouds and hybrid clouds. For enterprises that place a high value on data confidentiality and computational integrity, outsourcing data and computation may not be a justifiable risk. Private cloud solutions address this need, with services being accessible only within enterprise intranets, with limited benefits of cloud computing such as on-demand scaling and load balancing, as well as added initial infrastructure and set-up costs to new services. At the other end of the spectrum are the public clouds, where large third-party owned server farms and data centres host client data and computation services for hire. Two concerns stand out here: *insider attacks*, where employees within the public cloud enterprise can observe and infer trade secrets, and *side channels*, e.g., inadvertent or indirect information leaks when data belonging to different companies who may be competitors share the same bare metal. Analysis of side channels is outside the scope of this paper. A third model, the hybrid cloud model adopts a best of both worlds where depending on the sensitivity of the data and computation, enterprises may choose to offload only a part of their services on public clouds.

Our Secure k -NN solution is specifically targeted at public clouds. In particular, we work in what is called the two-party federated cloud setting, with two non-clustering public cloud servers, which is introduced in Twin Clouds [11] and subsequently used by the current state-of-the-art secure k -NN solution [14]. Federated clouds are an example of what are called interclouds [20], a collection of global stand-alone clouds. Interclouds allow better

load balancing and allocation of resources and help in addressing specific scenarios, e.g., services that are region-specific and require the data to be stored in a particular geography. Coordination of services across the intercloud can be centralized, or peer-to-peer as in the case of federated clouds. A detailed survey of the taxonomy of intercloud architectures is presented in [20].

2.2 Trust Assumptions

With federated clouds, as with public clouds, the trust model is of an *honest-but-curious* or the semi-honest adversary, who does not tamper directly with the computation or data. However, the adversary is free to observe inputs and outputs, as well as side-effects of computation and other behavioural characteristics on the cloud networks and servers. This type of adversary is different from the passive observer or the malicious adversarial model traditionally considered in the past, as the adversary is trusted to perform the computation correctly, but in addition has access to the internal state of the service, which includes client data. Such an adversary can observe the state of memory, and network traffic, or study operating systems behaviour in response to client queries. The choice of this particular type of adversary is justified, as data owners have to relinquish this control to the cloud service providers. While these exposures can be limited and controlled by legal contracts and liabilities, the threat of a curious insider can never be ruled out. This is where computing directly over encrypted data fills the gap. The goal is that even though we operate in the honest-but-curious adversary model, a malicious insider cannot obtain meaningful information from the data or the computation by observation. Further, we want to prevent the adversary from learning database access patterns, such as the set of (encrypted) result tuples returned corresponding to a particular input query, as well as the search patterns of queries, which may reveal information such as how many times the same query was issued.

Given an honest-but-curious adversary, the task of computing the k -nearest neighbours (k -NN) of a given query using a public cloud server, with the objective of achieving the above security goals in the cloud environment is difficult. One of the most attractive solutions is to use homomorphic encryption (HE). FHE (fully homomorphic encryption) [16] schemes allow the computation of arbitrary functions on encrypted data. Using FHE, data owners can offload their encrypted data to the public cloud, keeping all private or symmetric keys secret. When a client sends a query, the query is also encrypted using the same secret key and sent over to the cloud server. All computation on the cloud server is done on encrypted data and the resulting encrypted result is sent back to the clients, without compromising data confidentiality or the integrity of computation, even in the honest-but-curious adversary setting. Note that in this model, however, the actual algorithm, which is the sequence of computational steps on the data, is known to the insider, and is considered public. Care must be taken to design the algorithm to prevent leakage of search patterns and access patterns. Knowledge of the algorithm should not reveal anything about the data, query or results. Such FHE schemes however come at a significant operational cost and are not currently practical. For example, the state-of-the-art homomorphic sorting algorithm [12], takes 2 minutes to just sort 64 data items (32 bit), whereas as we show in our paper in Section 5, we are able to compute k -NN securely, with $k = 2$ for 30000 real-world data points (each point having 23 dimensions) in the same time.

¹<http://archive.ics.uci.edu/ml/>

To address performance issues of FHE, somewhat and partial HE ((S)HE and PHE) schemes have been proposed, which allow a restricted set of operations or sequences of operations on encrypted data, and are therefore more efficient. Examples include Paillier encryption [27] (which allow encrypted additions), LFHE [9], and BGN [8] (which allow computing restricted depth functions on encrypted data). While using simpler and more efficient schemes with restricted functionality, only a part of the computation is performed directly on the encrypted data. When more complex calculations are called for, these have to be done on plaintext values of intermediate results. To do this, another cloud provider is incorporated in the protocol. This cloud service provider is also assumed to be honest-but-curious and has access to secret keys which will allow it to decrypt the result of partial computations. Using intermediate plaintext values, more complicated operations are performed on these partial results, re-encrypted and sent back to the original cloud. The two clouds do not collude, the assumption is justified as these are public servers governed by legal contracts, and may even be business rivals. Colluding with each other will affect their reputation. This setting is called the *federated cloud model* as explained earlier.

The challenge now is to show that knowledge of such intermediate results does not leak information about the original data, query, and results as well as the access and search patterns. In this context, the state-of-the-art algorithm designed by Yousef et al. [14], shows that it is possible to design a secure k -NN scheme, where some of the secrets are given to one of the cloud servers, but both cloud servers do not learn anything about the original data. Our work also takes advantage of this model but explores a novel construction using a (S)HE scheme, and a more efficient protocol, allowing for asymptotically more efficient implementations for real-world scenarios.

2.3 Encrypted Computation

To find k -Nearest Neighbours, the distance between a given query point and all the other points in the database needs to be computed (in some appropriate dimension and measure). For computing Euclidean distance say in a two-dimensional space i.e., between two points (x_1, y_1) and (x_2, y_2) we need to compute $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$. In the interest of efficiency of computation, we can avoid the square root operation and work with squared Euclidean distances. This computation requires subtraction, squaring and addition operations in that order. After we compute the squared Euclidean distances in the cloud, we need to find the k minimum values. This requires us to order encrypted values.

Order Preserving Encryption (OPE), first proposed in [7] is also emerging as an important area of new research. Using OPE, it is possible for an observer to compute the order between two ciphertexts without having to decrypt them. Any OPE solution therefore necessarily leaks the original plaintext order among the encrypted ciphertexts [2], even if it does not reveal the value of the plaintexts themselves. OPE solutions by themselves are not sufficient for our problem, since our definition of Secure k -NN does not allow the adversary to learn anything about the original points or the query, and points have to be in plaintext for the server to order them after computing the distances. In [40], it is shown that solving Secure k -NN in the single cloud model securely implies that a secure OPE solution exists in this context, one which does not leak any information, including the very order, which is leaked by definition, and this is not possible.

(S)HE schemes on the other hand offer more functionality (rather than simple OPE) on encrypted data, and can be used cleverly to build a secure k -NN scheme.

2.4 Homomorphic Encryption and the LFHE scheme

We now give a brief description of the salient features of HE schemes and highlight the features of the leveled FHE (LFHE) [9] scheme we used in our implementation. HE allows direct computations to be performed on encrypted values, giving an encrypted result, which when decrypted gives the same plaintext results as if the same computations were performed on the plaintext values. An FHE scheme allows the computation of arbitrary functions on encrypted data.

A public key FHE scheme is an ensemble of four polynomial time algorithms:

- $(sk, pk) \leftarrow \text{KeyGen}(\$)$, the generation of a random public key and corresponding secret key pair.
- $c \leftarrow \text{Enc}_{pk}(m)$, the encryption of the a message m with the public key pk to produce a ciphertext c .
- $m \leftarrow \text{Dec}_{sk}(c)$, the decryption of the ciphertext with secret key sk to produce the same plaintext m .
- $c' \leftarrow \text{Eval}_{pk}(\phi, c_1, \dots, c_n)$ where ϕ is an arbitrary function in the message space, c_1, \dots, c_n are encryptions of inputs m_1, \dots, m_n to function ϕ . The result c' is the encryption of $m' = \phi(m_1, \dots, m_n)$, the encrypted output of application of the function on the plaintext inputs.

A partial homomorphic encryption scheme PHE is an HE scheme with a pre-defined function ϕ' , a restriction on the arbitrary function allowed in FHE. This restriction can be one function, such as addition or a sequence of functions in order, leading to a somewhat homomorphic encryption scheme ((S)HE) such as BGN, or a restriction on the kind of function (degree of the polynomial it can evaluate), for performance reasons. We assume that PHE and (S)HE satisfy the standard notion of security, whether it is against chosen plaintext attacks, which guarantee that ciphertexts output by the chosen HE scheme are indistinguishable from random, even to an adversary with access to an encryption oracle.

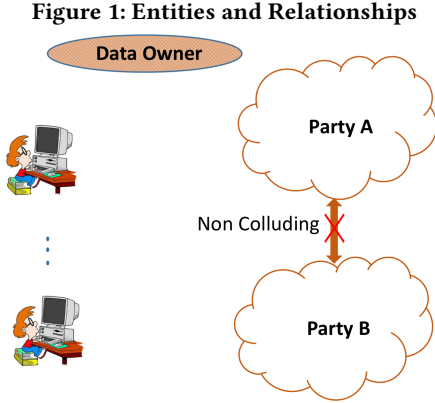
The (S)HE scheme we use in this paper is the Levelled FHE (LFHE) scheme implemented in HELib [17]. This LFHE scheme is based on the Brakerski-Gentry-Vaikuntanathan (BGV) scheme[10], and includes optimizations to make homomorphic evaluation runs faster, focusing mostly on the effective use of the Smart-Vercauteren [33] ciphertext packing techniques and the Gentry-Halevi-Smart [18] optimizations, with support for bootstrapping. The scheme is based on the ring learning with errors (RLWE) and the LWE problem, which have 2^λ security against known attacks. In addition to the **KeyGen**, **Enc**, and **Dec** functions, picking a level in the implementation L picks a depth L arithmetic circuit in **Eval**, with the computation quasilinear in the security parameter λ , lower levels corresponding to lower overheads. Further details of the LFHE scheme are presented in Appendix A.

3 SECURE k -NN

In this Section, we now describe our new Secure k -NN protocol in the non-colluding two-party setting described in Section 2. Section 3.1 presents the entities involved and Section 3.2 lays down the notation. Section 3.3 gives protocol details. The security guarantees of our protocol are discussed in detail in Section 4

and the performance characteristics are explored in detail in Section 5.

3.1 Entities



The main entities in our protocol are shown in Figure: 1.

- **Data owner** : The data owner is a trusted entity, the legal owner of the plaintext database, who outsources the storage of the encrypted data to Party A for k -NN computation, i.e., the plaintext database is not revealed to the party A. Once the data is outsourced, the data owner can be offline.
- **Party A** : Party A implements the storage and computation on an encrypted database. It receives encrypted query inputs and responds with k encrypted database points that represent the k -NN of the query point. It does not have access to any secret keys and works only on the encrypted data. Party A is assumed to be an honest-but-curious adversary, who will not tamper with the normal execution of the protocol, but has access to the internal state of its implementation, and could attempt to infer additional information about the characteristics of the plaintext data, query, results, or access and search patterns.
- **Party B** : Party B has access to the secret keys used to encrypt a given database, and does not have access to the encrypted database or query directly. Instead, it only has access to (partial) results of computations done on the encrypted data. Similar to Party A, Party B is honest-but-curious and does not tamper with the computation. However, it can use any information provided to infer characteristics about the plaintext database and query. A Secure k -NN solution will show that this information cannot be learned by Party B even if it has the secret key.
- **Clients** : These are users who are authorised by the data owner to interface with Party A and ask k -NN queries on the outsourced database. Clients have access to keys which allow them to send encrypted queries and decrypt the corresponding responses.

As the name suggests, in the non-colluding two party setting, Party A and Party B do not collude to expose the plaintext database to each other.

3.2 Notation

Our database P consists of n points $p_1, p_2 \dots p_n$. Each point is d -dimensional. The query Q is also a d -dimensional point.

- $D(p_i, Q)$ represents the squared Euclidean distance between p_i and Q .
- (S)HE: is a somewhat homomorphic encryption scheme which allows computation of this distance measure in the encrypted domain itself. For the distance measure used in this paper i.e. Euclidean Distance, we use the LFHE scheme [9]. Depending on the level chosen, e.g., with level 2, we can compute other measures such as Manhattan distance etc.

3.3 Setup

Figure 2 shows the entities, the communication and the computation phases in our main protocol. The Setup phase is executed once at the time of transferring the encrypted data to Party A. Let $(sk, pk) \leftarrow \text{KeyGen}(\$)$ be the secret-key and public-key respectively for the chosen (S)HE.

- Party A receives the public key pk from the data owner, as shown in label 1 in Figure 2.
- Party A also receives the encrypted database P' from the data owner, where the magnitude of each dimension d of each point p_i is encrypted under (S)HE using Enc_{pk} . These d encrypted values together constitute the d dimensional encrypted data points p'_i .
- Party B receives both sk and pk , as shown in label 2 in Figure 2.
- Clients receive both sk and pk , as shown in label 3 in Figure 2.

The inputs to our Secure k -NN protocol are the n encrypted points in P' and an encrypted d -dimensional query point $Q' \leftarrow \text{Enc}_{pk}(Q)$, of the plaintext query point Q computed by the client as shown in label 4 in Figure 2.

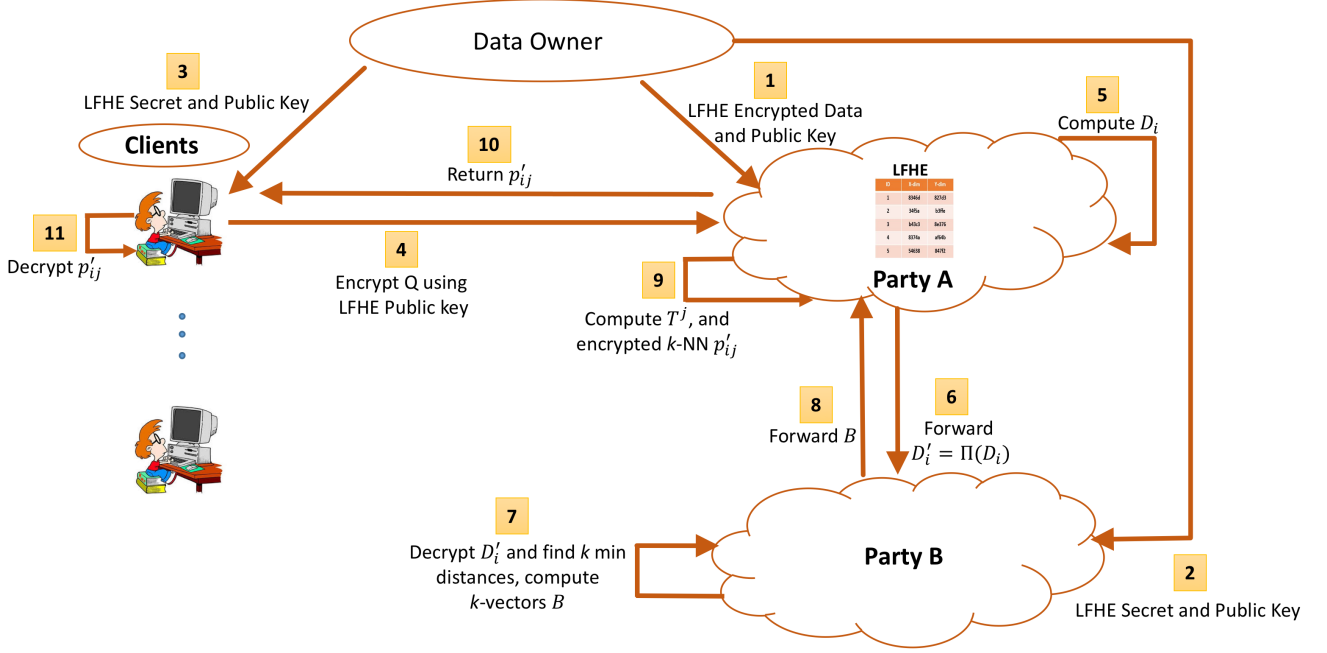
The output of the protocol, returned by Party A to the client, is the set of k encrypted points $\langle p'_{i_1}, p'_{i_2}, \dots, p'_{i_k} \rangle$, which are the encrypted k nearest neighbours to query Q in database P .

3.4 Secure k -NN Protocol

We describe our Secure k -NN protocol, which involves only one round of communication between the two parties, alternating between computations performed at Party A and Party B. At Party A, the computations are performed on encrypted data, and we show how no information is revealed to a curious insider apart from the protocol parameters, which are inputs to the algorithm itself. Party B has the secret key, but only sees values that are obtained by applying some function derived from the points in the database. We present the security guarantees of our protocol in detail in Section 4.

Compute Distances: In the first phase of our protocol after Setup at Party A, as shown in label 5 in Figure 2, we are given the encrypted database P' , and encrypted query Q' . For each d -dimensional point $p'_i \in P'$, i.e., the for all the n points in the encrypted database, we find the squared Euclidean distance between the given point and the query using our (S)HE with Eval_{pk} as shown in Steps 2–4 in Algorithm 1. Each ED_i is the encrypted value of the (square of the) distance between the given query and the i -th database point. In Step 5, we pick a polynomial $m(x)$ of the form $a_0 + a_1 \cdot x + a_2 \cdot x^2 + \dots + a_p \cdot x^p$ for some random $p \in \mathbb{N}$, where the coefficients a_0, \dots, a_p are picked uniformly at random within the range of the values of the (S)HE domain. For example, in our prototype implementation presented in this paper, we use positive random values picked from the range of points in $[1, 2^{32} - 1]$. We evaluate this monotonically increasing

Figure 2: Secure k -NN Protocol



1 Compute Distances: Party A

Data: P', Q'

Result: D'_i , the D_i in random permuted order

```

1 begin
2   for Each  $p'_i \in P'$  do
3      $ED_i \leftarrow \text{FindEncryptedDistance}(p'_i, Q')$ 
4     FindEncryptedDistance uses  $\text{Eval}_{pk}$  to compute
        $\sum_{i=1}^d (p'_i - Q'_i)^2$ 
5     Pick  $m$  a monotonically increasing polynomial with
       random coefficients
6     for  $i \leftarrow 1$  to  $n$  do
7        $D_i \leftarrow \text{EvalPoly}(m, ED_i)$ 
8       EvalPoly uses  $\text{Eval}_{pk}$  to evaluate  $m$  with  $ED_i$ 
9     Pick permutation  $\Pi$  over  $i$  points
10    Send  $D'_i \leftarrow \Pi(D_i)$  to Party B

```

polynomial transformation on each encrypted database point using **EvalPoly** to obtain the D_i s in Steps 6–8. Finally, we pick a permutation Π uniformly at random over the $n(n-1)/2$ points D_i s as $D'_i \leftarrow \Pi(D_i)$, in Steps 9–10 and send the distances to Party B out of order, as shown in label 6 in Figure 2.

Find Neighbours: In the second phase of our protocol at Party B, as shown in label 7 in Figure 2, the transformed points D'_i are received in random permuted order. We now form two vectors (arrays) of size k : NN and $NNindex$. In Steps 2–5 in Algorithm 2, we initialize $NN[i]$ to the first k points by decrypting the D'_i s (since we have the secret key sk), and store the corresponding index values $(1, \dots, k)$ in $NNindex$ as shown. For the remaining $k-n$ points, in steps 7–10, we first find the point with the maximum distance in NN and its corresponding $maxindex$ as

2 Find Neighbours: Party B

Data: D'_i , the D_i in random permuted order

Result: $\mathcal{B} = B^{i1}, \dots, B^{ik}$

```

1 begin
2    $NNindex[i] \leftarrow 0$ 
3   for  $i \leftarrow 1$  to  $k$  do
4      $NN[i] \leftarrow \text{Dec}_{sk}(D'_i)$ 
5      $NNindex[i] = i$ 
6   for  $i \leftarrow k+1$  to  $n$  do
7      $max \leftarrow NN[1], maxindex \leftarrow 1;$ 
8     for  $j \leftarrow 2$  to  $k$  do
9       if  $NN[j] > max$  then
10         $max \leftarrow NN[j]; maxindex \leftarrow j$ 
11     if  $(d \leftarrow \text{Dec}_{sk}(D'_i)) < max$  then
12        $NN[maxindex] = d$ 
13        $NNindex[maxindex] = i$ 
14    $\mathcal{B} \leftarrow \emptyset$ 
15   for  $j \leftarrow 1$  to  $k$  do
16     for  $i \leftarrow 1$  to  $n$  do
17       if  $i = NNindex[j]$  then
18          $B_i^j \leftarrow \text{Enc}_{pk}(1)$ 
19       else
20          $B_i^j \leftarrow \text{Enc}_{pk}(0)$ 
21      $\mathcal{B} \leftarrow \mathcal{B} \cup B^j$ 
22   Send  $\mathcal{B}$  to Party A

```

shown. Next, in Steps 11–13, for the new point in the i th position, we decrypt the value D'_i and check if it is smaller than the max distance we have seen so far. If it is, then we replace

the maximum value in NN with this and update the index value appropriately. At the end of this outer loop, the array NN will contain the k smallest values and the corresponding $NNIndex$ will track the index of these points. It is easy to see that the points corresponding to these permuted indices will be the k nearest neighbours for the query point in the original database, however, we need to transfer this information to Party A without revealing either the permuted indices or these values directly.

To do this, we construct the set \mathcal{B} of k row vectors B^j , where each B^j is a n -vector. For each permuted index value in $NNIndex$, we populate the array B^j as follows: for the position i in B^j at the $NNIndex[j]$ value, i.e., $i = NNIndex[j]$ we store an encryption of the value 1 (Party B also has the public key pk), and in the remaining $n - 1$ i -positions we store an encryption of the value 0 as shown in Steps 15–21. This new vector B^j is added to the set \mathcal{B} . We repeat this k times until all indices in $NNIndex$ have been processed. At the end of this phase, as shown in label 8 in Figure 2, we send the k row vectors \mathcal{B} to Party A (Step 22 in Algorithm 2).

3 Return kNN: Party A

Data: $\mathcal{B} \leftarrow B^1, \dots, B^k$
Result: $p'_{i1}, p'_{i2}, \dots, p'_{ik}$

```

1 begin
2   Receive set of  $k$   $n$ -dimensional vectors  $\mathcal{B}$ 
3   for  $j \leftarrow 1$  to  $k$  do
4      $T^j \leftarrow (\Pi(P^j)) \cdot B^j$ 
5      $p'_{ij} \leftarrow \text{Enc}(0)$ 
6     for  $l \leftarrow 1$  to  $n$  do
7        $p'_{ij} \leftarrow p'_{ij} + T_l^j$ 
8   Return  $p'_{i1}, p'_{i2}, \dots, p'_{ik}$ 

```

Return kNN: In the next phase of our protocol, Party A receives k n -dimensional row vectors (Algorithm 3 Step 2). As shown in label 9 of Figure 2, and Step 4 of the algorithm, we first apply the permutation Π selected by Party A in Algorithm 1 to the encrypted database points P^j . Applying the permutation Π on the points corrects the order of the 0s and 1s in B^j that was derived from the permuted sums in Party B. Next, we compute the scalar dot product (pointwise multiplication) between this permuted row vector with each of our B^j s, giving us an n -dimensional vector T^j . After this multiplication, the original point that is one of the k -NNs, i.e., p'_{ij} , will remain (in encrypted form) in T^j , all other entries will become (encrypted) zeros. We sum the n elements of each of the T^j row vectors as shown in Steps 5–7. The k vector sums are the encrypted k -NNs, i.e., the points $p'_{i1}, p'_{i2}, \dots, p'_{ik}$, which can now be returned as the final result in Step 8 of algorithm and label 10 in Figure 2. Clients can decrypt these p'_{ij} s using sk to obtain the plaintext points that correspond to the k nearest neighbours as shown in label 9 in Figure 2.

There are two main *novel* ideas in our protocol:

- The use of m , a monotonically increasing polynomial with uniformly random coefficients, which effectively masks the database values to Party B.
- The construction of vectors B^j along with the uniformly random permutation Π , which prevents Party A from learning the query results, as well as the access pattern and query search pattern.

The communication and computation overheads in our protocol are as follows. Party A computes $O(n)$ encrypted values, and sends one set of n values to Party B. Party B computes n decryptions and $O(nk)$ encryptions and returns $O(nk)$ encrypted values to Party A. In the last phase Party A performs $O(nkd)$ operations on the ciphertexts.

3.5 Comparison of Performance and Efficiency with Yousef et al.

We present an algorithmic comparison of the efficiency of our protocol with that of Yousef et al. [14], the current state-of-the-art scheme for Secure k -NN in the two-party model. Both schemes involve a pair of non-colluding parties, one of which is in possession of the secret key of the somewhat homomorphic encryption scheme underlying the respective protocol. The first major advantage of our protocol is the reduced number of round communications - one, as compared to k for Yousef et al., where k is the number of nearest neighbours requested by the client. Thus, the cost of round communication in their protocol depends on the query, while in our protocol it is constant. Additionally, the construction of Yousef et al. performs bit-level decomposition operations on the encrypted data, which are costly and require specific capabilities which only certain (S)HE schemes can afford (e.g. Yousef et al. use the Paillier cryptosystem-based (S)HE). On the other hand, our protocol only exploits the basic capabilities that any (S)HE scheme can afford. In other words, our protocol uses the (S)HE scheme as a black-box, which can be easily instantiated using known (S)HE schemes such as LFHE (leveled fully homomorphic encryption) with optimized implementations. The comparison in terms of computational overheads is presented in Table 1 for computing k -NN over n data points, each of dimension d such that each dimension takes l bit values and m is a degree D polynomial in our solution.

4 SECURITY GUARANTEES

In this section, we discuss the security of our protocol in detail. We begin with the assumptions. In our secure k -NN protocol, Party A and Party B are assumed to be honest-but-curious, as described earlier. Party A will follow the protocol steps correctly, but we cannot rule out insider attacks. As discussed earlier, we do not address side channels in this paper. We also assume Party A and Party B do not collude. Party B additionally is trusted with the secret key of the (S)HE scheme. Party B again is honest-but-curious, and anything that Party B has or can compute is also assumed to be exposed to an insider. Informally, our protocol is secure in the following sense: even if this key is compromised and the data values exposed to an insider in Party B, the original database, query, and results are still secret to both Party A and B. For emphasis, we recall that Party B and Party A do not collude, and Party A does not have the decryption keys.

We present our security arguments focusing on the views of the two untrusted parties - Party A and Party B, under the assumption that they are mutually non-colluding. All other parties (clients and database owners) are trusted.

4.1 Leakage Profile for Party A

We explicitly enumerate the leakage to Party A at different phases of the overall protocol. Observe that Party A is involved in **Compute Distances** and **Return kNN**, we focus on the leakage to Party A during each of these:

Table 1: Computational overheads

	Yousef et al	Our Secure k -NN protocol
Number of Homomorphic operations	$O(n(2kl + d))$	$O(n(k + d + D))$
Number of encryptions	$O(nkl)$	$O(nk)$
Number of decryptions (Party B)	$O(n(kl + d))$	$O(n)$
Number of round communications	$O(k)$	1
Communication overhead per round	$O(nl + d)$ bits	$O(nl)$ bits

- Compute Distances:** In this phase, Party A computes the set of encrypted distances ED_i between each encrypted data point p_i and the encrypted query point Q' (Steps 3–4). Each of these computations is performed using the **FindEncryptedDistance** function, which in turn uses the homomorphic evaluation function **Eval** of the (S)HE scheme. Hence, the CPA security guarantees of the underlying (S)HE ensures that none of these computations reveal any information about the underlying data points or the plaintext distances to Party A. Subsequently, Party A homomorphically evaluates a randomly chosen polynomial m on each ED_i using the **EvalPoly** function to obtain the corresponding encrypted output D_i . Again, since all polynomial evaluations are on encrypted data, any non-negligible leakage to Party A from these computations amounts to a violation of the CPA security of the (S)HE scheme.
- Return kNN:** This phase is an oblivious transfer of the knowledge of the k -nearest data points to Party A. Specifically, we argue that Party A does not learn which k points in the encrypted database correspond to the output set of points p'_{i1}, \dots, p'_{ik} . The first observation that we use is that the set of k n -dimensional vectors B^1, \dots, B^k received by Party A contain encryptions of 0 and 1. As already mentioned, the CPA security guarantees of the underlying (S)HE ensures that Party A cannot distinguish between these 0 and 1 entries. Hence the vectors themselves do not reveal to Party A their correspondence to the respective points in the database. The inner product computations in Steps 4–7 are again performed homomorphically and leak no information to Party A. Finally, observe that multiplying an encrypted point by an encryption of 1 essentially results in a randomized re-encryption of the same point. Since, Party A has no knowledge of which entry in the vectors B^1, \dots, B^k corresponds to 1, the output points p'_{i1}, \dots, p'_{ik} cannot be traced back to the originally encrypted points in the database. Once again, any violation of the above guarantee amounts to a violation of the CPA security of the (S)HE scheme.

The aforementioned leakage profile for Party A leads to the following security guarantee with respect to Party A:

THEOREM 4.1. Secure k -NN Guarantee: Party A: *Our secure k -nearest neighbour protocol leaks no information to Party A except the number of nearest neighbours k returned by the protocol. In particular, Party A gains no knowledge of the access pattern, that is, the set of points in the database corresponding to the k -nearest neighbour points returned by the protocol, and does not learn the query pattern, which reveals if two queries were the same.*

4.2 Leakage Profile for Party B

In this section, we explicitly enumerate the leakage to Party B in the **Find Neighbours** phase of the protocol. Party B receives

the set D'_i of encrypted polynomial evaluation outputs $m(ED_i)$ in random permuted order. Note that since we have picked a secure pseudorandom permutation, which is computationally difficult to invert, implying that the exact identity of points associated with any given difference value is hidden from Party B. Since the polynomial m is order preserving, Party B can sort the decrypted polynomial outputs. We now examine the possibility of any leakage to Party B from the resulting system of ordered equations. Let $d''_1 < d''_2 < \dots < d''_n$ be the ordered set of plaintext distances, and $m(d''_1) < m(d''_2) < \dots < m(d''_n)$ be the ordered set of polynomial outputs obtained by Party B upon decryption. As mentioned earlier, the polynomial $m(x)$ is of the form $a_0 + a_1 \cdot x + a_2 \cdot x^2 + \dots + a_p \cdot x^p$ for some random $p \in \mathbb{N}$. Party B can formulate the following system of equations:

$$\begin{aligned}
 m(d''_1) &= a_0 + a_1 \cdot d''_1 + a_2 \cdot (d''_1)^2 + \dots + a_n \cdot (d''_1)^p \\
 m(d''_2) &= a_0 + a_1 \cdot d''_2 + a_2 \cdot (d''_2)^2 + \dots + a_n \cdot (d''_2)^p \\
 &\vdots \\
 m(d''_n) &= a_0 + a_1 \cdot d''_n + a_2 \cdot (d''_n)^2 + \dots + a_n \cdot (d''_n)^p
 \end{aligned}$$

where only the left-hand side of each equation is known to Party B. Without loss of generality, we may assume that Party B can guess with high probability the degree p of the polynomial chosen by Party A, as well as the range of values (say $[0, 2^N]$) that each plaintext distance d''_i can take. This is a particularly relevant assumption in the context of real-world datasets, where the adversary may possess some apriori knowledge of the range of Euclidean distances between the data points. In addition, since homomorphic polynomial evaluation in the encrypted domain is a costly operation, the degree p can only take a small range of values, which Party B can also accurately guess in a small number of trials. However, we prove that even if Party B has full knowledge of the aforementioned parameters, it cannot recover the original data points within a feasible amount of computation time. Observe that the system of equations has exactly $n + p + 1$ unknown variables from Party B's point of view, while the number of equations is only n . Hence, Party B must correctly guess the $p + 1$ smallest distances $d''_1, d''_2, \dots, d''_{p+1}$ to recover the polynomial coefficients. The average number of possible values that these distances can take is $\binom{2^N}{p+1}$, which is approximately the same as $2^{N \cdot (p+1)}$ for $2^N \gg (p + 1)$. In other words, the probability that Party B successfully recovers the polynomial coefficients, and subsequently the plaintext distances, is approximately $1/2^{N \cdot (p+1)}$, which is close to negligible. For example, for $N = 16$ and $p = 9$, the probability that Party B is able to recover the plaintext distances is approximately 2^{-160} , which is close to negligible for a security level of 160 bits. Thus even when the range of plaintext distances and the degree of the polynomial chosen by Party A are reasonably small and known apriori to Party B, the information leakage is negligible. Also note that Party A refreshes the polynomial for each query point, implying that Party B gains no

additional information across the queries. Finally, even if Party B is able to recover the plaintext distances in some extreme cases (e.g., when the plaintext distance values follow some specific pattern), it still does not directly reveal the plaintext data points to Party B, as the query point is also unknown.

The only possible leakage to Party B in this round is the presence of such points in the database that are equidistant from the query point Q . This is leaked from the presence of identical values in the set of values $m(d''_1), m(d''_2), \dots, m(d''_n)$. However, since the order of the values is randomly permuted by Party A, Party B cannot map these values back to the original index of the data points in the database.

The aforementioned leakage profile for Party B leads to the following security guarantee with respect to Party B:

THEOREM 4.2. Secure k -NN Guarantee: Party B: *Our secure k -nearest neighbour protocol leaks no information to Party B except the number of nearest neighbours k to be returned by the protocol and the number of equidistant points in the database with respect to a given query point Q .*

4.3 Comparison of Security Guarantees with Yousef et al.

We conclude the discussion on the security of our k -NN protocol with a comparison of our security guarantees with those afforded by the current state-of-the-art scheme proposed by Yousef et al. [14]. Both schemes involve a pair of non-colluding parties, one of which is in possession of the secret key of the somewhat homomorphic encryption scheme underlying the respective protocol. In both schemes, one of the parties (Party A in our case), encounters only encrypted data and consequently, learns nothing about the data access pattern or search pattern from the protocol. In Yousef et al.'s protocol, one of the parties (equivalent to Party B) learns the distance between the query point and the global nearest neighbour at each stage of the protocol. Moreover, the presence of an irreversible permutation implies that this leakage cannot be mapped back to an actual point in the database. This distance value is not leaked by our protocol.

In our protocol, Party B learns the presence of pairwise equidistant points in the database with respect to a given query point. That is, whether there are two or more points that are equidistant is revealed to Party B, not the value of the distances unlike in Yousef et al. Moreover, the pairwise distances in our protocol are randomly permuted, which ensures that these leakages cannot be mapped back to identify the corresponding point pairs in the actual database. In both protocols, in spite of the knowledge of some distance values, or the knowledge of the number of equidistant points, the original database, query, results or access and search patterns cannot be deduced.

In the next section, we describe our reference implementation and show how it is asymptotically faster than the state-of-the-art algorithm and present its performance characteristics on simulated and real-world data sets.

5 IMPLEMENTATION AND EXPERIMENTS

In this section, we present our implementation details and record the experiments carried out to test the performance of the proposed protocol. Our experimental setup consists of four machines, representing the data owner, Party A, Party B and client. The configuration of machines representing Party A and Party B are: 4 core 2.8 GHz processors, 16 GB RAM running Ubuntu 16.04

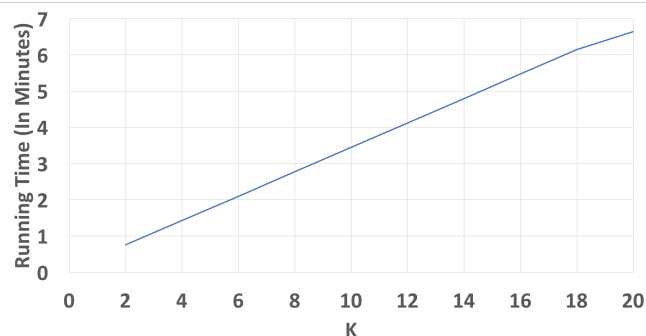


Figure 3: Running time for real world cancer data set, 858 points with 32 dimensions

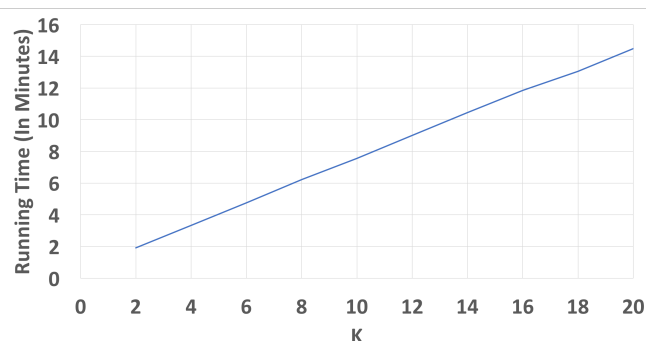


Figure 4: Running time for real world credit card data set, 30000 points with 23 dimensions

LTS. The configuration of machines representing Data Owner and Client are: 4 core 2.8 GHz processors, 8 GB RAM running Ubuntu 16.04 LTS.

We used the HELib [17] library, with LFHE as the underlying encryption mechanism. This library is written in C++ and we implemented our protocols also in C++. In our implementation, we set $p = 1099511627689$, a large prime between 2^{40} and 2^{87} , the maximum depth to 10 and the security parameter to 128, i.e., offering 2^{128} bits of security.

5.1 Real world data

Our first set of results is on real-world data from the UCI Machine learning repository [24]. We focus on two datasets: Cervical cancer (Risk Factors) and Default of credit card clients. Our goal is to test how our Secure k -NN algorithm performs when we attempt to find reports that cluster near a chosen query report. The cervical cancer dataset contains 858 data points each having 32 dimensions, representing demographic information, habits, and historic medical records of 858 patients. The default of credit card clients dataset contains 30000 data points each having 23 dimensions including sensitive information such as the amount of given credit, gender, age, education, marital status etc. We pre-processed these datasets so that they contain only non-negative integer values. Both these datasets have PII information and are good candidates for encrypted analytics. Again we emphasize that our goal in this study is not to comment on the predictive accuracy of k -NN by itself as a suitable data mining algorithm for these datasets.

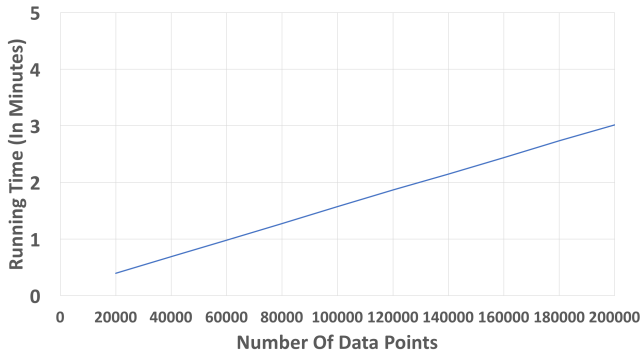


Figure 5: Running time when number of dimensions = 2 and $k = 5$

Figure 3 shows the measured running time of our protocol when we vary k from 2 to 20 on the cancer data set. As shown in the figure, we can compute 2-NN for all the points in less than a minute (45 s), 8-NN in 2 minutes and 45 seconds and 16-NN in 5 minutes and 28 seconds. The function grows linearly with k .

Figure 4 shows the measured running time of our protocol when we vary k from 2 to 20 for the credit card clients data. As shown in the figure, we can compute 2-NN in less than 2 minutes and 20-NN in 14 minutes and 20 seconds, with the function growing linearly in k . In each of these experiments, we generate a random data point to serve as the query point. The experiment is then repeated with multiple such query points and the average time taken to execute a query is recorded. From the figures, it is clear that our protocol scales linearly with k .

Both these experiments show that our implementation overheads are very competitive, making a case for real-world application of our algorithms to perform simple pattern matching on sensitive data. Other applications where this technique can be directly applied include spatial databases and location-based search (for example a taxi-for-hire application), where the query looks for points within a small set of records that are already filtered.

Note that in our protocol, the two parties A and B communicate with each other directly, without going back to the client. The communication cost between the parties, i.e., sending the monotonically increasing function of squared Euclidean distances, is independent of the number of dimensions in the original dataset, i.e., only one value is sent for each pair of points (regardless of the value of d). Response from Party B is also independent of the dimension of the data. The computational overhead of calculating the squared Euclidean distances on Party A depends naturally on the dimensions, which is unavoidable.

5.2 Simulation

There are three varying parameters in our protocol, a) number of data points n b) the number of dimensions d and c) k the number of nearest neighbours. By generating synthetic data we are able to keep two parameters fixed and check the effect of the third parameter on running time of the query. Our simulation-data generator uses a uniform random distribution to generate the data.

Figure 5 shows how the running time varies when the number of data points are increased, keeping the number of dimensions d and k constant (5 in this case). From this figure, we can clearly

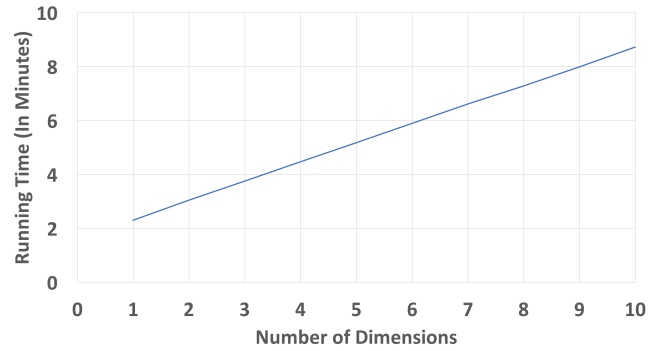


Figure 6: Running time when number of data points = 200000 and $k = 2$

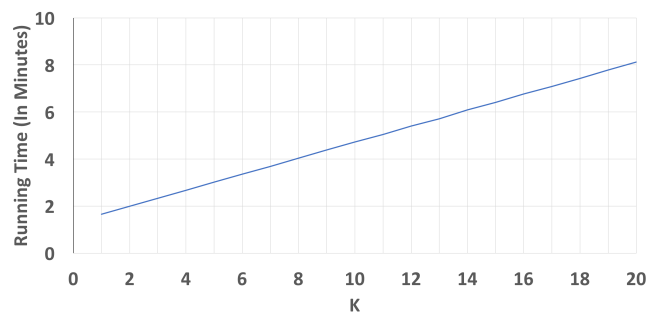


Figure 7: Running time when number of data points = 200000 and dimensions = 2

see that the running time of our protocol ranges from 23 seconds up to 3 minutes, and grows linearly with the size of the dataset n , which ranges from 20000 to 200000 points.

Figure 6 shows how the running time varies when the number of dimensions in the data is increased, from 1 to 10, keeping the number of data points n as 200000 and $k = 2$ as constant. The running time ranges from 2 minutes and 17 seconds to less than 9 minutes for 10 dimensions. The figure clearly shows that our protocol scales linearly with the number of dimensions.

Figure 7 shows how the running time varies when k is increased from 1 to 20 keeping the number of data points as 200000 and number of dimensions $d = 2$. The running time ranges from less than 2 minutes to around 8 minutes as shown. The figure clearly shows that the running time scales linearly with k .

From the above experiments, it is clear that the running time of our protocol is linear in k , n and d . Note that our protocol requires only **one** round of communication between Party A and Party B. In comparison, Yousef et al., requires at least $O(k)$ rounds of communication to compute the secure k -NN, with each round sending $O(nb)$ points where n is the database size and b the bit size of every element. Additionally, more rounds are required to compute the squared Euclidean distances and secure bit decomposition, which are used subsequently to compute k -NN.

For a similar machine configuration², for 2000 points and 6 dimensions, with $k = 25$ our protocol runs in 1 minute and 37 seconds, whereas Yousef et al., report a running time of 55 minutes and 39 seconds. We emphasize that while this comparison is

²6 cores, 3.07 GHz processor and 12GB RAM running Ubuntu 10.04 LTS

done on different machines, the trend observed is explained by our efficient one-round communication and the simplified computations on Parties A and B, without compromising security guarantees.

6 RELATED WORK

Finding k - Nearest Neighbours is a fundamental operation in many data mining and machine learning algorithms. In this section, we present an overview of the techniques developed for secure k -nearest neighbour problem (SkNN). SkNN is a specific use case of the much broader problem of secure processing over outsourced encrypted data. This area has been well studied and many techniques have been proposed. Some of the techniques are generic, i.e. they are able to support a wide variety of queries over the encrypted data, while others enable a particular operation, such as k -NN, over the encrypted data. The existing work can be categorized into various buckets depending on the underlying mechanism used, including privacy preserving data mining, garbled circuit implementations of secure k -NN, secure multiparty computation (SMC), private information retrieval (PIR) schemes, secure hardware-based techniques and other solutions that directly implement secure k -NN protocols.

The first three solutions, privacy-preserving data mining, garbled circuits, and SMC, cannot be compared directly with our work, as the goals, assumptions and models are different as discussed next. In privacy-preserving data mining [3] [15] [23] [25] [35] privacy is achieved by transforming the data using anonymization models such as k -anonymity, data perturbation i.e adding noise to the data, suppressing some tuples, etc. The transformed data preserves the ability to answer the required query within an accepted error limit. The drawback of these techniques is that they lead to information loss. Typically a superset of the results are returned and the client has to process the query results further. These techniques outsource the transformed data to the server in plaintext, which leaks information as well.

Songhori et al[34] propose to use Yao's garbled circuit protocol for secure k -NN computation. In their setting, Alice has a query point Q and Bob has the dataset S . They want to jointly compute the k -NN of Q in S such that Bob does not learn anything about Q and Alice does not learn anything about S except k -NN. This solution is not geared to outsourcing as Bob has the data in plaintext.

Secure multiparty computation (SMC) techniques enable multiple parties to securely evaluate a function of their private inputs without revealing the inputs of one party to the others. SMC has been leveraged to compute SkNN by various solutions [32] [31] [37]. They partition the data and give to multiple parties, which then compute k -NN as an SMC. Again, this is fundamentally different from our work because the parties can view the data in plaintext.

Providing secure database-as-a-service has been investigated by CryptDB[30] and Monomi [36]. They use property preserving encryption such as OPE [2][7][6] to handle a portion of the query processing directly on encrypted data stored at the cloud efficiently. The portion of query which cannot be processed by these property preserving encryption schemes is executed at the client side after decryption of data returned by cloud. Monomi [36] shows that such solutions can provide good performance if they are tuned properly for the query workload, though for an arbitrary query the performance can deteriorate significantly.

Also, it needs a full-fledged database engine at the client side which is not always possible, e.g., on mobile devices.

Secure Hardware based techniques, including Cipherbase and others[1] [26] [4] [5] assume the availability of a secure co-processor at the cloud server. These co-processors are specially built such that no outside entity can read the data stored inside it (tamper-proof hardware). The computations performed inside the co-processor are also isolated from the outside world. Data owners upload the keys to the co-processor in a secure manner. The logic for processing, for example, k -NN, is also installed on the co-processor. For query processing, the encrypted data and encrypted query point are sent to the co-processor, which decrypts the data using the keys stored on it and runs the installed logic on the decrypted data returning the final answer in encrypted form to the client. Such techniques give strong security guarantees. However, the co-processors are resource constrained and are less powerful than the regular processors with limited memory available to them. Also, the deployment and maintenance of such co-processors is not straight forward, as some operations have to be performed by the clients, e.g., key refresh, requiring the client to maintain the hardware.

Using PIR (Private Information retrieval) for secure k -NN has been studied in [19] [29]. PIR allows users to retrieve an object X_i from a set $X = X_1, X_2, \dots, X_n$ stored at the server without revealing i to server. Again, PIR schemes work on plaintext data on the server and guarantee that a user's query point will not be revealed, different from our setting.

Wong et al[39] propose a new encryption scheme called ASPE (Asymmetric Scalar Product Preserving Encryption), to compare the distance between a query point and compute the distances required for k -NN. Hu et al[21] propose a secure k -NN method based on provable secure homomorphic encryption. However, the setting used by them is different from us in that the client has the ciphertext while the server has the capability to decrypt. Both solutions are vulnerable to Chosen Plaintext Attacks [40].

Yao et al[40] establish a relationship between the SkNN problem and the order preserving encryption (OPE) problem. They show that SkNN is at least as hard as OPE in a single cloud setting. They propose a solution based on Voronoi diagrams, in which the server returns a superset of results for the k -NN. On the other hand, we return the exact result of k -NN to the client in the two party federated cloud setting.

Sunoh et al[13] provide a solution for secure k -NN which uses mOPE(mutable Order Preserving Encryption). They propose two methods, one based on Voronoi diagrams, which returns the exact k -NNs but is expensive when $k > 1$, and another method based on triangulation, which is more efficient but gives exact results for only $k = 1$. For $k > 1$ it gives false positives which have to be filtered out by the client. Their solution also requires round communication between the server and client to first reduce the potential candidates and to find the k -NN. No security proofs are provided. Also, this solution has an expensive database update procedure which requires changing of encrypted data.

In his seminal work, Gentry [16] proposed a construction for a fully homomorphic encryption scheme (FHE). FHE allows computation of any function directly on encrypted data and a solution for computing k -NN can be developed using this. As discussed earlier, the computational cost of FHE is very high for real world applications.

Yousef et al[14] describe the current state-of-the-art protocol for SkNN in the two party federated cloud model. They use Paillier encryption [27] as the underlying cryptographic tool. Paillier

encryption is additive homomorphic. Using this property they develop protocols which compute k -NN securely in the federated cloud model. Their solution provides very strong security guarantees and is able to hide the data access pattern as well. However, this security comes at the cost of performance, taking minutes to perform queries that we can execute in seconds and milliseconds. We are able to improve upon the performance characteristics as shown in Section 5, while maintaining strong security guarantees, making it more suitable for real-world deployments.

Other recent related work includes Lei et al. [22], a SkNN scheme for 2-D data points using LSH (location sensitive hashing). They first construct the secure index for the data and then outsource the secure index and encrypted data to the cloud. Since the scheme uses LSH data structures, their results contain false positives. Also related to computing on encrypted data, though not k NN, is Seabed [28], where the authors use an additively symmetric homomorphic encryption scheme to compute large-scale aggregations of data in an enterprise setting and prevent against frequency attacks.

7 CONCLUSIONS

This paper describes a new protocol for efficient secure k - Nearest Neighbours on encrypted data. We use LFHE, a somewhat homomorphic encryption scheme as our basic building block, in the two party federated cloud model. Our adversary model captures insider attacks under the honest-but-curious assumption. Our protocol is fast compared to the state of the art, without compromising on security guarantees. Our implementations are fast and scalable, and our experiments on real-world data show how basic data mining on encrypted data can be practical. In the future, we plan to extend our work to other data mining algorithms, including k -Means and Apriori.

REFERENCES

- [1] R. Agrawal, D. Asonov, M. Kantarcioglu, and Yaping Li. 2006. Sovereign Joins. In *22nd International Conference on Data Engineering (ICDE'06)*. 26–26. <https://doi.org/10.1109/ICDE.2006.144>
- [2] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. 2004. Order preserving encryption for numeric data. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. ACM, 563–574.
- [3] Rakesh Agrawal and Ramakrishnan Srikant. 2000. Privacy-preserving Data Mining. *SIGMOD Rec.* 29, 2 (May 2000), 439–450. <https://doi.org/10.1145/335191.335438>
- [4] Arvind Arasu, Spyros Blanas, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravi Ramamurthy, and Ramaratnam Venkatesan. 2013. Orthogonal security with cipherbase. In *Proc. of the 6th CIDR, Asilomar, CA*.
- [5] S. Bajaj and R. Sion. 2014. TrustedDB: A Trusted Hardware-Based Database with Privacy and Data Confidentiality. *IEEE Transactions on Knowledge and Data Engineering* 26, 3 (March 2014), 752–765. <https://doi.org/10.1109/TKDE.2013.38>
- [6] Alexandra Boldyreva, Nathan Chenette, Younho Lee, and Adam O’neill. 2009. Order-preserving symmetric encryption. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 224–241.
- [7] Alexandra Boldyreva, Nathan Chenette, and Adam O’Neill. 2011. Order-preserving encryption revisited: Improved security analysis and alternative solutions. In *Annual Cryptology Conference*. Springer, 578–595.
- [8] Dan Boneh, Eu-Jin Goh, and Kobbi Nissim. 2005. Evaluating 2-DNF Formulas on Ciphertexts. In *Proceedings of the Second International Conference on Theory of Cryptography (TCC'05)*. Springer-Verlag, Berlin, Heidelberg, 325–341. https://doi.org/10.1007/978-3-540-30576-7_18
- [9] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2012. (Leveled) Fully Homomorphic Encryption Without Bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference (ITCS '12)*. ACM, New York, NY, USA, 309–325. <https://doi.org/10.1145/2090236.2090262>
- [10] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2014. (Leveled) Fully Homomorphic Encryption Without Bootstrapping. *ACM Trans. Comput. Theory* 6, 3, Article 13 (July 2014), 36 pages. <https://doi.org/10.1145/2633600>
- [11] Sven Bugiel, Stefan Nürnberg, Ahmad-Reza Sadeghi, and Thomas Schneider. 2011. Twin Clouds: Secure Cloud Computing with Low Latency. In *Proceedings of the 12th IFIP TC 6/TC 11 International Conference on Communications and Multimedia Security (CMS'11)*. 32–44.

- [12] Gizem S. Çetin, Yarkin Doröz, Berk Sunar, and Erkey Savas. 2015. Low Depth Circuits for Efficient Homomorphic Sorting. *IACR Cryptology ePrint Archive* 2015 (2015), 274.
- [13] S. Choi, G. Ghinita, H. S. Lim, and E. Bertino. 2014. Secure kNN Query Processing in Untrusted Cloud Environments. *IEEE Transactions on Knowledge and Data Engineering* 26, 11 (Nov 2014), 2818–2831. <https://doi.org/10.1109/TKDE.2014.2302434>
- [14] Y. Elmehdwi, B. K. Samanthula, and W. Jiang. 2014. Secure k-nearest neighbor query over encrypted data in outsourced environments. In *2014 IEEE 30th International Conference on Data Engineering*. 664–675. <https://doi.org/10.1109/ICDE.2014.6816690>
- [15] Alexandre Evfimievski, Ramakrishnan Srikant, Rakesh Agrawal, and Johannes Gehrke. 2002. Privacy Preserving Mining of Association Rules. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '02)*. ACM, New York, NY, USA, 217–228. <https://doi.org/10.1145/775047.775080>
- [16] Craig Gentry. 2009. *A Fully Homomorphic Encryption Scheme*. Ph.D. Dissertation. Stanford, CA, USA. Advisor(s) Boneh, Dan. AAI3382729.
- [17] Craig Gentry and Shai Halevi. 2011. Implementing Gentry’s Fully-homomorphic Encryption Scheme. In *Proceedings of the 30th Annual International Conference on Theory and Applications of Cryptographic Techniques: Advances in Cryptology (EUROCRYPT'11)*. Springer-Verlag, Berlin, Heidelberg, 129–148. <http://dl.acm.org/citation.cfm?id=2008684.2008697>
- [18] Craig Gentry, Shai Halevi, and Nigel P. Smart. 2012. Better Bootstrapping in Fully Homomorphic Encryption. In *Proceedings of the 15th International Conference on Practice and Theory in Public Key Cryptography (PKC'12)*. Springer-Verlag, Berlin, Heidelberg, 1–16. https://doi.org/10.1007/978-3-642-30057-8_1
- [19] Gabriel Ghinita, Panos Kalnis, Ali Khoshgozaran, Cyrus Shahabi, and Kian-Lee Tan. 2008. Private Queries in Location Based Services: Anonymizers Are Not Necessary. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08)*. ACM, New York, NY, USA, 121–132. <https://doi.org/10.1145/1376616.1376631>
- [20] Nikolay Grozev and Rajkumar Buyya. 2014. Inter-Cloud architectures and application brokering: taxonomy and survey. *Software: Practice and Experience* 44, 3 (2014).
- [21] Haibo Hu, Jianliang Xu, Chushi Ren, and Byron Choi. 2011. Processing Private Queries over Untrusted Data Cloud Through Privacy Homomorphism. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering (ICDE '11)*. IEEE Computer Society, Washington, DC, USA, 601–612. <https://doi.org/10.1109/ICDE.2011.5767862>
- [22] X. Lei, A. X. Liu, and R. Li. 2017. Secure KNN Queries over Encrypted Data: Dimensionality Is Not Always a Curse. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. 231–234. <https://doi.org/10.1109/ICDE.2017.91>
- [23] Ninghui Li, Tiancheng Li, and Suresh Venkatasubramanian. 2007. t-closeness: Privacy beyond k-anonymity and l-diversity. In *ICDE 2007*. IEEE, 106–115.
- [24] M. Lichman. 2017. UCI Machine Learning Repository. <http://archive.ics.uci.edu/ml>
- [25] Ashwin Machanavajjhala, Daniel Kifer, Johannes Gehrke, and Muthuramkrishnan Venkatasubramanian. 2007. l-diversity: Privacy beyond k-anonymity. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 1, 1 (2007), 3.
- [26] E. Mykletun and G. Tsudik. 2005. Incorporating a secure coprocessor in the database-as-a-service model. In *Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA'05)*. 7 pp.–. <https://doi.org/10.1109/IWIA.2005.28>
- [27] Pascal Paillier. 1999. *Public-Key Cryptosystems Based on Composite Degree Residuosity Classes*. Springer Berlin Heidelberg, Berlin, Heidelberg, 223–238. https://doi.org/10.1007/3-540-48910-X_16
- [28] Antonis Papadimitriou, Ranjita Bhagwan, Nishanth Chandran, Ramachandran Ramjee, Andreas Haeberlen, Harmeet Singh, Abhishek Modi, and Saikrishna Badrinayanan. 2016. Big data analytics over encrypted datasets with seabed. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, 587–602.
- [29] Stavros Papadopoulos, Spiridon Bakiras, and Dimitris Papadias. 2010. Nearest Neighbor Search with Strong Location Privacy. *Proc. VLDB Endow.* 3, 1-2 (Sept. 2010), 619–629. <https://doi.org/10.14778/1920841.1920920>
- [30] Raluca Ada Popa, Catherine M. S. Redfield, Nikolai Zeldovich, and Hari Balakrishnan. 2011. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. ACM, New York, NY, USA, 85–100. <https://doi.org/10.1145/2043556.2043566>
- [31] Yinian Qi and Mikhail J Atallah. 2008. Efficient privacy-preserving k-nearest neighbor search. In *Distributed Computing Systems, 2008. ICDCS'08. The 28th International Conference on*. IEEE, 311–319.
- [32] Mark Shaneck, Yongdae Kim, and Vipin Kumar. 2006. Privacy preserving nearest neighbor search. In *Data Mining Workshops, 2006. ICDM Workshops 2006. Sixth IEEE International Conference on*. IEEE, 541–545.
- [33] N. P. Smart and F. Vercauteren. 2010. Fully Homomorphic Encryption with Relatively Small Key and Ciphertext Sizes. In *Proceedings of the 13th International Conference on Practice and Theory in Public Key Cryptography (PKC'10)*. Springer-Verlag, Berlin, Heidelberg, 420–443. https://doi.org/10.1007/978-3-642-13013-7_25
- [34] Ebrahim M. Songhori, Siam U. Hussain, Ahmad-Reza Sadeghi, and Farinaz Koushanfar. 2015. Compacting Privacy-preserving K-nearest Neighbor Search Using Logic Synthesis. In *Proceedings of the 52Nd Annual Design Automation*

- Conference (DAC '15). ACM, New York, NY, USA, Article 36, 6 pages. <https://doi.org/10.1145/2744769.2744808>
- [35] Latanya Sweeney. 2002. k-anonymity: A model for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 10, 05 (2002), 557–570.
- [36] Stephen Tu, M. Frans Kaashoek, Samuel Madden, and Nikolai Zeldovich. 2013. Processing Analytical Queries over Encrypted Data. *Proc. VLDB Endow.* 6, 5 (March 2013), 289–300. <https://doi.org/10.14778/2535573.2488336>
- [37] Jaideep Vaidya and Chris Clifton. 2005. Privacy-preserving top-k queries. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*. IEEE, 545–546.
- [38] V. Vaikuntanathan. 2011. Computing Blindfolded: New Developments in Fully Homomorphic Encryption. In *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*. 5–16. <https://doi.org/10.1109/FOCS.2011.98>
- [39] Wai Kit Wong, David Wai-lok Cheung, Ben Kao, and Nikos Mamoulis. 2009. Secure kNN Computation on Encrypted Databases. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data (SIGMOD '09)*. ACM, New York, NY, USA, 139–152. <https://doi.org/10.1145/1559845.1559862>
- [40] Xiaokui Xiao, Feifei Li, and Bin Yao. 2013. Secure Nearest Neighbor Revisited. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013) (ICDE '13)*. IEEE Computer Society, Washington, DC, USA, 733–744. <https://doi.org/10.1109/ICDE.2013.6544870>
- [41] Youwen Zhu, Zhiqiu Huang, and Tsuyoshi Takagi. 2016. Secure and controllable -NN query over encrypted cloud data with key confidentiality. *J. Parallel and Distrib. Comput.* 89 (2016), 1 – 12. <https://doi.org/10.1016/j.jpdc.2015.11.004>

A THE LFHE SYSTEM

The general encryption of BGV scheme that can be instantiated to both LWE and RLWE. We will describe RLWE which used by HELib. The RLWE-based public key encryption scheme as follows. Most of the description and equations are taken from[9] [38].

In general, homomorphic encryption scheme is a tuple (HE.KeyGen, HE.Enc, HE.Dec, HE.Eval) of probabilistic polynomial time algorithms. In BGV, the message space of the scheme will always be some ring \mathbf{RM} and our computational model will be arithmetic circuits over this ring (i.e. addition and multiplication gates).

- (1) HE.KeyGen takes the security parameter (and possibly other parameters of the scheme) and produces a secret key sk and a public key pk .
- (2) HE.Enc takes the public key pk a message m and produces a ciphertext c , which is the encryption of m .
- (3) HE.Dec takes the secret key sk and a ciphertext c and produces a message m .
- (4) HE.Eval takes the public key pk , an arithmetic circuit f over \mathbf{RM} , and ciphertexts c_1, \dots, c_l where l is the number of inputs to f , and outputs a ciphertext c_f .

Given the security parameter λ and an additional parameter μ , first choose a μ -bit modulus q . Where q an odd positive modulus $q = q(\lambda)$. For RLWE scheme, chose the degree $d = d(\lambda, \mu)$, a "noise" distribution $\chi = \chi(\lambda, \mu)$, let the "dimension" $n = \lceil 3 \log q \rceil$. Let $R_q = \mathbb{Z}_q[x]/(f(x))$ with $f(x)$ a polynomial of degree d . $f(x) = x^d + 1$ and $d = d(\lambda)$ is a power of 2. To get the secret key, first draw \mathbf{s}' uniformly from χ . The secret key is then

$$\mathbf{s} = (1, \mathbf{s}') \in R_q^2$$

To get the public key, first generate vectors $\mathbf{A}' \leftarrow R_q^n$, $e \leftarrow \chi^n$, then set $b = -\mathbf{A}'\mathbf{s}' + 2e$. Set public key $A = (b|\mathbf{A}') \in R_q^{n \times 2}$. Note that $A \cdot \mathbf{s} = 2e$.

Suppose $m \in \{0, 1\}$ is the bit we wanting to encrypt. To encrypt, we do the following:

- (1) Select a random $r \in R_q^n$ and expand the message $m = (m, 0) \in R_q^n$.
- (2) Output $= m + A^T r \in R_q^n$.

According to $RLWE_{d,q,\chi}$ where χ is a uniform distribution over R_q , we can use this scheme a polynomial number of times with negligible probability that an adversary can guess \mathbf{s} .

To decrypt, do the following:

- (1) Compute $b' = \lfloor \langle \mathbf{c}, \mathbf{s} \rangle \rfloor_q$
- (2) Output $m = \lfloor b' \rfloor_2$

A Parallel and Scalable Processor for JSON Data

Christina Pavlopoulou
University of California, Riverside
cpavl001@ucr.edu

E. Preston Carman, Jr
University of California, Riverside
ecarm002@ucr.edu

Till Westmann
Couchbase
tillw@apache.org

Michael J. Carey
University of California, Irvine
mjcarey@ics.uci.edu

Vassilis J. Tsotras
University of California, Riverside
tsotras@cs.ucr.edu

ABSTRACT

Increasing interest in JSON data has created a need for its efficient processing. Although JSON is a simple data exchange format, its querying is not always effective, especially in the case of large repositories of data. This work aims to integrate the JSONiq extension to the XQuery language specification into an existing query processor (Apache VXQuery) to enable it to query JSON data in parallel. VXQuery is built on top of Hyracks (a framework that generates parallel jobs) and Algebricks (a language-agnostic query algebra toolbox) and can process data on the fly, in contrast to other well-known systems which need to load data first. Thus, the extra cost of data loading is eliminated. In this paper, we implement three categories of rewrite rules which exploit the features of the above platforms to efficiently handle path expressions along with introducing intra-query parallelism. We evaluate our implementation using a large (803GB) dataset of sensor readings. Our results show that the proposed rewrite rules lead to efficient and scalable parallel processing of JSON data.

1 INTRODUCTION

The Internet of Things (IoT) has enabled physical devices, buildings, vehicles, smart phones and other items to communicate and exchange information in an unprecedented way. Sophisticated data interchange formats have made this possible by leveraging their simple designs to enable low overhead communication between different platforms. Initially developed to support efficient data exchange for web-based services, JSON has become one of the most widely used formats evolving beyond its original specification. It has emerged as an alternative to the XML format due to its simplicity and better performance [28]. It has been used frequently for data gathering [22], motion monitoring [20], and in data mining applications [24].

When it comes time to query a large repository of JSON data, it is imperative to have a scalable system to access and process the data in parallel. In the past there has been some work on building JSONiq add-on processors to enhance relational database systems, e.g. Zorba [2]. However, those systems are optimized for single-node processing.

More recently, parallel approaches to support JSON data have appeared in systems like MongoDB [10] and Spark [7]. Nevertheless, these systems prefer to first load the JSON data and transform them to their internal data model formats. On the other hand systems like Sinew [29] and Dremel [27] cannot query raw JSON data. They need a pre-processing phase to convert the input file into a readable binary for them (typically Parquet [3]). They can then load the data, transform it to their internal data model

and proceed with its further processing. The above efforts are examples of systems that can process JSON data by converting it to their data format, either automatically, during the loading phase, or manually, following the pre-processing phase. In contrast, our JSONiq processor can immediately process its JSON input data without any loading or pre-processing phases. Loading large data files is a significant burden for the overall system's execution time as our results will show in the experimental section. Although, for some data, the loading phase takes place only in the beginning of the whole processing, in most real-time applications, it can be a repetitive action; data files to be queried may not always be known in advance or they may be updated continuously.

Instead of building a JSONiq parallel query processor from scratch, given the similarities between JSON and XQuery, we decided to take advantage of Apache VXQuery [4, 17], an existing processor that was built for parallel and scalable XQuery processing. We chose to support the JSONiq extension to XQuery language [8] to provide the ability to process JSON data. XQuery and JSONiq have certain syntax conflicts that need to be resolved for a processor to support both of them, so we enhanced VXQuery with the *JSONiq extension* to the XQuery language, an alteration of the initial JSONiq language designed to resolve the aforementioned conflicts [9].

In extending Apache VXQuery, we introduce three categories of JSONiq rewrite rules (*path expression*, *pipelining*, and *group-by* rules) to enable parallelism via pipelining and to minimize the required memory footprint. A useful by-product of this work is that the proposed group-by rules turn out to apply to both XML and JSON data querying.

Through experimentation, we show that the VXQuery processor augmented with our JSONiq rewrite rules can indeed query JSON data without adding the overhead of the loading phase used by most of the state-of-the-art systems.

The rest of the paper is organized as follows: Section 2 presents the existing work on JSON query processing, while Section 3 outlines the architecture of Apache VXQuery. Section 4 introduces the specific optimizations applied to JSON queries and how they have been integrated into the current version of VXQuery. The experimental evaluation appears in Section 5. Section 6 concludes the paper and presents directions for future research.

2 RELATED WORK

Previous work on querying data interchange formats has primarily focused on XML data [26]. Nevertheless there has been considerable work for querying JSON data. One of the most popular JSONiq processors is Zorba [2]. This system is basically a virtual machine for query processing. It processes both XML and JSON data by using the XQuery and JSONiq languages respectively. However, it is not optimized to scale onto multiple nodes with multiple data files, which is the focus of our work. In

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

contrast, Apache VXQuery is a system that can be deployed on a multi-node cluster to exploit parallelism.

A few parallel approaches for JSON data querying have emerged as well. These systems can be divided into two categories. The first category includes SQL-like systems such as Jaql [14], Trill [18], Drill [6], Postgres-XL [11], MongoDB [10] and Spark [13], which can process raw JSON data. Specifically, they have been integrated with well-known JSON parsers like Jackson [1]. While the parser reads raw JSON data, it converts it to an internal (table-like) data model. Once the JSON file is in a tabular format, it can then be processed by queries. Our system can also read raw JSON data, but it has the advantage that it does not require data conversion to another format since it directly supports JSON’s data model. Queries can thus be processed on the fly as the JSON file is read. It is also worthwhile mentioning that Postgres-XL (a scalable extension to PostgreSQL [12]) has a limitation on how it exploits its parallelism feature. Specifically, while it scales on multiple nodes it is not designed to scale on multiple cores. On the other hand, our system can be multinode and multicore at the same time. In the experimental section we show how our system compares with two representatives from this category (MongoDB and Spark).

We note that AsterixDB [5], can process JSON data in two ways. It can either first load the file internally (like the systems above) or, it can access the file as external data without the need of loading it. However, in both cases and in contrast to our system, AsterixDB needs to convert the data to its internal ADM data model. In our experiments we compare VXQuery with both variations of AsterixDB.

Systems in the second category (e.g. Sinew [29], Argo [19] and Oracle’s system [25]) cannot process raw JSON data and thus need an additional pre-processing phase (hence an extra overhead than the systems above). During that phase, a JSON file is converted to a binary or Parquet ([3]) file that is then fed to the system for further transformation to its internal data model before query processing can start.

Systems like Spark and Argo process their data in-memory. Thus, their input data sizes are limited by a machine’s memory size. Recently, [23] presents an approach that pushes the filters of a given query down into the JSON parser (Mison). Using data-parallel algorithms, like SIMD vectorization and Bitwise Parallelism, along with speculation, data not relevant to the actual query is filtered out early. This approach has been added into Spark and improves its JSON performance. Our work also prunes irrelevant data, but does so by applying rewrite rules. Since the Mison code is not available yet, we could not compare with them in detail; we also need to note that Mison is just a parallel JSON parser for JSON data. In contrast, VXQuery is an integrated processor that can handle the querying of both JSON and XML data (regardless of how complex the query is).

As opposed to the aforementioned systems, our work builds a new JSONiq processor that leverages the architecture of an existing query engine and achieves high parallelism and scalability via the employment of rewrite rules.

3 APACHE VXQUERY

Apache VXQuery was built as a query processing engine for XML data implemented in Java. It is built on top of two other frameworks, namely the Hyracks platform and the Algebricks layer. Figure 1, also, shows AsterixDB [5], which uses the same infrastructure.

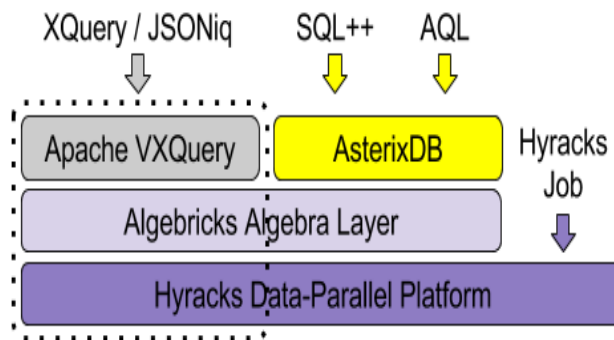


Figure 1: The VXQuery Architecture

3.1 Infrastructure

The first layer is *Hyracks* [16], which is an abstract framework responsible for executing dataflow jobs in parallel. The processor operating on top of Hyracks is responsible for providing the partitioning scheme while Hyracks decides how the resulting job will be distributed. Hyracks processes data in partitions of contiguous bytes, moving data in fixed-sized frames that contain physical records, and it defines interfaces that allow users of the platform to specify the data-type details for comparing, hashing, serializing and de-serializing data. Hyracks provides built-in base data types to support storing data on local partitions or when building higher level data types.

The next layer, *Algebricks* [15], takes as input a logical query plan and, via built-in optimization rules that it provides, converts it to a physical plan. Apart from the transformation, the rules are responsible for making the query plan more efficient. In order to achieve this efficiency, Algebricks allows the processor above (in this case Apache VXQuery) to provide its own language specific rewrite rules.

The final layer, *Apache VXQuery* [4, 17], supports a XQuery processor engine. To build a JSONiq processor, we used the JSONiq extension to XQuery specifications. Specifically, we focused mostly on implementing all the necessary modules to successfully parse and evaluate JSONiq queries. Additionally, several modules were implemented to enable JSON file parsing and support an internal in-memory representation of the corresponding JSON items.

The resulting JSONiq processor accepts as input the original query, in string form, and converts it to an abstract syntax tree (AST) through its query parser. Then, the AST is transformed with the help of VXQuery’s translator to a logical plan, which becomes the input to Algebricks.

As mentioned above, VXQuery uses Hyracks to schedule and run data parallel jobs. However, Hyracks is a data-agnostic platform, while VXQuery is language-specific. This creates a need for additional rewrite rules to exploit Hyracks’ parallel properties for JSONiq. If care is not taken, the memory footprint for processing large JSON files can be prohibitively high. This can make it impossible for systems with limited memory resources to efficiently support JSON data processing. In order to identify opportunities for parallelism as well as to reduce the runtime memory footprint, we need to examine in more depth the characteristics of the JSON format as well as the supported query types.

3.2 Hyracks Operators

We first proceed with a short description of the Hyracks logical operators that we will use in our query plans.

- **EMPTY-TUPLE-SOURCE:** outputs an empty tuple used by other operators to initiate result production.
- **DATASCAN:** takes as input a tuple and a data source and extends the input tuple to produce tuples for each item in the source.
- **ASSIGN:** executes a scalar expression on a tuple and adds the result as a new field in the tuple.
- **AGGREGATE:** executes an aggregate expression to create a result tuple from a stream of input tuples. The result is held until all tuples are processed and then returned in a single tuple.
- **UNNEST:** executes an unnesting expression for each tuple to create a stream of output tuples per input.
- **SUBPLAN:** executes a nested plan for each tuple input. This plan consists of an AGGREGATE and an UNNEST operator.
- **GROUP-BY:** executes an aggregate expression to produce a tuple for each set of items having the same grouping key.

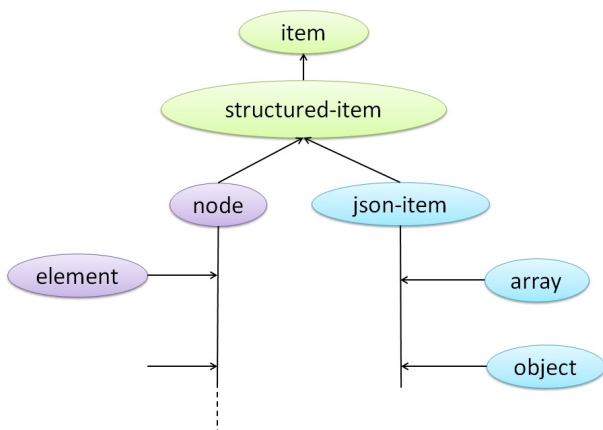


Figure 2: XML vs JSON structure

It is imperative for understanding this work to describe the representation along with the navigation expressions of JSON items according to the JSONiq extension to the XQuery specification. A json-item can be either an array or an object, in contrast to an XML structure, which consists of multiple nodes as described in Figure 2. An array consists of an ordered list of items (members), while an object consists of a set of pairs. Each pair is represented by a key and a value. The following is the terminology used for JSONiq navigation expressions:

- **Value:** for an array it yields the value of a specified (by an index) array element, while for an object it yields the value of a specified (by a field name) key.
- **Keys-or-members:** for an array it outputs all of its elements, and for an object it outputs all of its keys.

4 JSON QUERY OPTIMIZATION

The JSONiq rewrite rules are divided into three categories: the Path Expression, Pipelining, and Group-by Rules. The first category removes some unused expressions and operators, as well as

streamlining the remaining path expressions. The second category reduces the memory needs of the pipeline. The last category focuses on the management of aggregation, which also contains the group-by feature (added to VXQuery in the XQuery 3.0 specification). For all our examples, we will consider the bookstore structure example depicted in Listing 1.

```
{
  "bookstore": {
    "book": [
      {
        "-category": "COOKING",
        "title": "Everyday Italian",
        "author": "Giada De Laurentiis",
        "year": "2005",
        "price": "30.00"
      },
      ...
    ]
  }
}
```

Listing 1: Bookstore JSON File

4.1 Path Expression Rules

The goal of the first category of rules is to enable the unnesting property. This means that instead of creating a sequence of all the targeted items and processing the whole sequence, we want to process each item separately as it is found. This rule opens up opportunities for pipelining since each item is passed to the next stage of processing as the previous step is completed.

```
json-doc("books.json")("bookstore")("book")()
```

Listing 2: Bookstore query

The example query in Listing 2 asks for all the books appearing in the given file. Specifically, it reads data from the JSON file ("books.json") and then, the *value* expression is applied twice, once for the bookstore object ("bookstore") and once for the book object ("book"). In this way, it is ensured that only the matching objects of the file will be stored in memory. The value of the book object is an array, so the *keys-or-members* expression (()) applied to it returns all of its items. To process this expression, we first store in a tuple all of the objects from the array and then we iterate over each one of them. The result that is distributed at the end is each book object separately.

```
DISTRIBUTE-RESULT($S9)
UNNEST($S9:iterate($S8))
ASSIGN($S8:(keys-or-members($S2)))
ASSIGN($S2:value(value(json-doc(promote(data("books.json"),
string),"bookstore"),"book"))
EMPTY-TUPLE-SOURCE
```

Figure 3: Original Query Plan

In more detail, we can describe the aforementioned process in terms of a logical query plan that is returned from VXQuery (Figure 3). It follows a bottom-up flow, so the first operator in the query plan is the EMPTY-TUPLE-SOURCE leaf operator. The empty tuple is extended by the following ASSIGN operator, which consists of a *promote* and a *data* expression to ensure that the json-doc argument is a string. Also, the two *value* expressions inside it verify that only the book array will be stored in the tuple.

The next two operators depict the two steps of the processing of the *keys-or-members* expression. The first operator is an ASSIGN, which evaluates the expression to extend its input tuple. Since this expression is applied to an array, the returned tuple includes all of the objects inside the array. Then, the UNNEST operator applies an iterate expression to the tuple and returns a stream of tuples including each object from the array.

The final step according to the query plan is the distribution of each object returned from the UNNEST. From the analysis above, we can observe that there are opportunities to make the logical plan more efficient. Specifically, we observe that there is no need for two processing steps of *keys-or-members*.

Originally, the tuple with all the book objects produced by the *keys-or-members* expression flows into the UNNEST operator whose iterate expression will return each object in a separate tuple. Instead, we can merge the UNNEST with the *keys-or-members* expression. That way, each book object is returned immediately when it is found.

Finally, to further clean up our query plan, we can remove the *promote* and *data* expressions included in the first ASSIGN operator. The fully optimized logical plan is depicted in Figure 4.

```
DISTRIBUTE-RESULT($$13 )
UNNEST($$13:keys-or-members($$2))
ASSIGN($$2:value(value(json-doc("books.xml"),"bookstore"),"book")
EMPTY-TUPLE-SOURCE
```

Figure 4: Updated Query Plan

The new and more efficient plan opens up opportunities for pipelining since when a matching book object is found, it is immediately returned and, at the same time, passed to the next stage of the process.

4.2 Pipelining Rules

This type of rule builds on top of the previous rule set and considers the use of the DATASCAN operator along with the way to access partitioned-parallel data. The sample query that we use is depicted in Listing 3.

```
collection("/books")("bookstore")("book")()
```

Listing 3: Bookstore Collection Query

According to the query plan in Figure 5, the ASSIGN operator takes as input data source a collection of JSON files, through the *collection* expression. Then, UNNEST *iterate* iterates over the collection and outputs each single file. The two value expressions integrated into the second ASSIGN output a tuple source filled with all the book objects produced by the whole collection. The last step of the query plan (created in the previous section) is implemented by the *keys-or-members* expression of the UNNEST operator, which outputs each single object separately.

The input tuple source generated by the *collection* expression corresponds to all the files inside the collection. This does not help the execution time of the query, since the result tuple can be huge. Fortunately, Algebricks offers its DATASCAN operator, which is able to iterate over the collection and forwards to the next operator each file separately. To accomplish this procedure, DATASCAN replaces both the ASSIGN *collection* and the UNNEST *iterate*.

```
DISTRIBUTE-RESULT($$13 )
UNNEST($$13:keys-or-members($$6))
ASSIGN($$6:value(value($$4,"bookstore"),"book"))
UNNEST($$4:iterate($$2))
ASSIGN(collection("/books"), $$2)
EMPTY-TUPLE-SOURCE
```

Figure 5: Query Plan for a Collection

```
DISTRIBUTE-RESULT($$13 )
UNNEST($$13:keys-or-members($$4))
ASSIGN($$4:value(value($$2,"bookstore"),"book"))
DATASCAN(collection("/books"), $$2)
EMPTY-TUPLE-SOURCE
```

Figure 6: Introduction of DATASCAN

By enabling Algebricks' DATASCAN, apart from pipeline improvement, we also achieve partitioned parallelism. In Apache VXQuery, data is partitioned among the cluster nodes. Each node has a unique set of JSON files stored under the same directory specified in the *collection* expression. The Algebricks' physical plan optimizer uses these partitioned data property details to distribute the query execution. Adding these properties allows Apache VXQuery to achieve partitioned-parallel execution without any user-level parallel programming.

To further improve pipelining, we can produce even smaller tuples. Specifically, we extend the DATASCAN operator with a second argument (here it is the book array). This argument defines the tuple that will be forwarded to the next operator.

In the updated query plan (Figure 6), the newly inserted DATASCAN is followed by an ASSIGN operator. Inside it, the two *value* expressions populate the tuple source with all the book objects of the file fetched from DATASCAN. We can merge the value expressions with DATASCAN by adding a second argument to it. As a result, the output tuple, which was previously filled with each file, is now set to only have its book objects (Figure 7).

```
DISTRIBUTE-RESULT($$13 )
UNNEST($$13:keys-or-members($$4))
DATASCAN(collection("/books"), $$4, ("bookstore")("book"))
EMPTY-TUPLE-SOURCE
```

Figure 7: Merge *value* with DATASCAN Operator

At this point, we note that by building on the previous rule set, both the query's efficiency and the memory footprint can be further improved. In the query plan in Figure 7, DATASCAN *collection* is followed by an UNNEST whose *keys-or-members* expression outputs a single tuple for each book object of the input sequence.

```
DISTRIBUTE-RESULT($$4 )
DATASCAN(collection("/books"), $$4, ("bookstore")("book"))
EMPTY-TUPLE-SOURCE
```

Figure 8: Merge *keys-or-members* with Datascan Operator

This sequence of operators gives us the ability to merge DATASCAN with *keys-or-members* by extending its second argument.

Figure 8 shows this action, whose result is the fetching of even smaller tuples to the next stage of processing. Specifically, instead of storing in DATASCAN's output tuple a sequence of all the book objects of each file in the collection, we store only one object at a time. Thus, query's execution time is improved and Hyracks' dataflow frame size restriction is satisfied.

```
for $x in collection("/books")("bookstore")
("book")()
group by $author:=$x("author")
return count($x("title"))
```

Listing 4: Bookstore Count Query

4.3 Group-by Rules

The last category of rules can be applied to both XML and JSON queries since the group-by feature is part of both syntaxes. Group-by can activate rules enabling parallelism in aggregation queries.

```
for $x in collection("/books")("bookstore")
("book")()
group by $author:=$x("author")
return count(for $j in $x return $j("title"))
```

Listing 5: Bookstore Count Query (2nd form)

The example query that we will use to show how our rules affect aggregation queries is in Listings 4 and 5. Both forms of the query read data from a collection with book files, group them by author, and then return the number of books written by each author.

```
DISTRIBUTE-RESULT($$20 )
UNNEST($$20:iterate($$19))
ASSIGN($$19:count(value($$18,"title")))
ASSIGN($$18:treat(item,$$16))
GROUP-BY($$17:0->$$14){
  AGGREGATE($$16:create_sequence($$13))
}
ASSIGN($$14:data(value($$13,"author")))
DATASCAN(collection("/books"), $$13, ("bookstore")("book")())
EMPTY-TUPLE-SOURCE
```

Figure 9: Query Plan with Count Function

In Figure 9, the DATASCAN *collection* passes a tuple, for one book object at a time, to ASSIGN. The latter applies the *value* expression to acquire the author's name for the specific object. GROUP-BY accepts the tuple with the author's name (group-by key) and then its inner focus is applied (AGGREGATE) so that all the objects whose author field have the same value will be put in the same sequence.

At this point, ASSIGN *treat* appears; this ensures that the input expression has the designated type. So, our rule searches for the type returned from the sequence created from the AGGREGATE operator. If it is of type *item* which is the *treat* type argument, the whole *treat* expression can be safely removed. As a result, the whole ASSIGN can now be removed since it is a redundant operator (Figure 10).

After the former removal, GROUP-BY is followed by an ASSIGN *count* which calculates the number of book titles (*value* expression) generated by AGGREGATE *sequence*. According to the JSONiq extension to XQuery, *value* can be applied only on a JSON object or array. However, in our case the query plan applies

```
DISTRIBUTE-RESULT($$20 )
UNNEST($$20:iterate($$19))
ASSIGN($$19:count(value($$16,"title")))
GROUP-BY($$17:0->$$14){
  AGGREGATE($$16:create_sequence($$13))
}
ASSIGN($$14:data(value($$13,"author")))
DATASCAN(collection("/books"), $$13, ("bookstore")("book")())
EMPTY-TUPLE-SOURCE
```

Figure 10: Query Plan without *treat* Expression

value to a sequence, since GROUP-BY aggregates all the records having the same group-by key in a sequence. Thus, ("title") is applied on a sequence. To overcome this conflict, we convert the ASSIGN to a SUBPLAN operator (Figure 11). SUBPLAN's inner focus introduces an UNNEST *iterate* which iterates over AGGREGATE *sequence* and produces a single tuple for each item in the sequence. The inner focus of SUBPLAN finishes with an AGGREGATE along with a count function which incrementally calculates the number of tuples that UNNEST feeds it with.

```
DISTRIBUTE-RESULT($$20 )
UNNEST($$20:iterate($$19))
SUBPLAN{
  AGGREGATE($$19:count(value($$18,"title")))
  UNNEST($$18:iterate($$16))
}
GROUP-BY($$17:0->$$14){
  AGGREGATE($$16:create_sequence($$13))
}
ASSIGN($$14:data(value($$13,"author")))
DATASCAN(collection("/books"), $$13, ("bookstore")("book")())
EMPTY-TUPLE-SOURCE
```

Figure 11: Convert scalar to aggregation expression

This conversion not only helps resolving the aforementioned conflict but it also converts the scalar count function to an aggregate one. This means that instead of calculating count on a whole sequence, we can incrementally calculate it as each item of the sequence is fetched.

In Figure 11, GROUP-BY still creates a sequence in its inner focus, which is the input to SUBPLAN UNNEST. Instead, we can push the AGGREGATE operator of the SUBPLAN down to the GROUP-BY operator by replacing it (Figure 12). That way, we exploit the benefits of the aforementioned conversion and have the count function computed at the same time that each group is formed (without creating any sequences). Thus, the new plan is not only smaller (more efficient) but also keeps the pipeline granularity introduced in both of the previous rule sets.

At this point, it is interesting to look at the second format of the query in Listing 5. The for loop inside the count function conveniently forms a SUBPLAN operator right above the GROUP-BY in the original logical plan. This is exactly the query plan described in Figure 11, which means that in this case we can immediately push the AGGREGATE down to GROUP-BY, without any further transformations.


```

DISTRIBUTE-RESULT($$20 )
UNNEST($$20:iterate($$16))
GROUP-BY($$17:0->$$14){
  AGGREGATE($$16:count(value($$13,"title")))
}
ASSIGN($$14:data(value($$13, "author"))
DATASCAN( collection("/books"), $$13, ("bookstore")("book")())
EMPTY-TUPLE-SOURCE

```

Figure 12: Updated Query Plan with Count Function

Now that the count function is converted into an aggregate function, there is another rule introduced in [17] that can be activated to further improve the overall query performance. This rule supports Algebricks' two-step aggregation scheme, which means that each partition can calculate locally the count function on its data. Then, a central node can compute the final result using the data gathered from each partition. Thus, partitioned computation is enabled, which improves parallelism effectiveness.

The final query plan, produced after the application of all the former rules, calculates the count function at the same time that each grouping sequence is built as opposed to first building it and then processing the aggregation function.

5 EXPERIMENTAL EVALUATION

We have tested our rewrite rules by integrating them into the latest version of Apache VXQuery [4]. Each node has two dual-core AMD Opteron(tm) processors, 8GB of memory, and two 1TB hard drives. For the multi-node experiments we built a cluster of up to nine nodes of identical configuration. We used a real dataset with sensor data and a variety of queries, described below in Sections 5.1 and 5.2 respectively. We repeated each query five times. The reported query time is an average of the five runs. We first consider single-node experiments and include measurements for execution time (before and after applying our rewrite rules) and for speed-up. For the multi-node experiments we measure response time and scale-up over different numbers of nodes. We, also, include comparisons with Spark SQL and MongoDB that show that the overhead of their loading phase is non-negligible. Finally, we compare with AsterixDB which has the same infrastructure as our system; in particular we compare with two approaches, one that sees the input as an external dataset (depicted in the figures as AsterixDB) and one that first loads the file internally (depicted as AsterixDB(load)).

5.1 Dataset

The data used in our experiments are publicly available from the National Oceanic and Atmospheric Administration (NOAA) [21]. The Daily Global Historical Climatology Network (GHCN-Daily) dataset was originally in dly format and was converted to its equivalent NOAA web service JSON representation.

Listing 6 shows an example JSON sensor file structure. All records are wrapped into a JSON item, specifically array, called "root". Each member of the "root" array is an object item which contains the object "metadata" and the array "results". The latter stores various measurements organized in individual objects. A specific measurement includes the date, the data type (a description of the measurement, with typical values being TMIN, TMAX, WIND etc.), the station id where the measurement was taken, and the actual measurement value. The "count" object

included into the "metadata" denotes the number of measurements objects in the accompanying "results" array. Typically a "results" array contains measurements from a given station for one month (i.e. typically 30 measurements). A sensor file contains only one "root" array which may consist of several "results" (measurements from the same or different stations) accompanied by their "metadata".

Sensor file sizes vary from 10MB to 2GB. Each node holds a collection of sensor files; the size of the collection varies from 100MB to 803GB. The collection size is varied throughout the experiments and is cited explicitly for each experiment. In our experiments, we assume that the data has already been partitioned among the existing nodes.

```

{
  "root": [
    {
      "metadata": {
        "count":31,
      },
      "results": [
        {
          "date": "20132512T00:00",
          "dataType": "TMIN",
          "station": "GSW123006",
          "value": 4
        },
        ...
      ]
    },
    {
      "metadata": {
        "count":29,
      },
      "results": [
        {
          "date": "20142512T00:00",
          "dataType": "WIND",
          "station": "GSW957859",
          "value": 30
        },
        ...
      ]
    },
    ...
  ]
}

```

Listing 6: Example JSON Sensor File Structure

5.2 Query Types

We evaluated our newly implemented rewrite rules by evaluating different types of queries including selection (Q0, Q0b), aggregation (Q1, Q1b) and join-aggregation queries (Q2). The query description follows.

```

for $r in collection("/sensors")("root")("results")()
let $datetime := dateTime(data($r("date")))
where year-from-dateTime($datetime) ge 2003
and month-from-dateTime($datetime) eq 12
and day-from-dateTime($datetime) eq 25
return $r

```

Listing 7: Selection Query (Q0)

```

for $r in collection("/sensors")("root")("results")("date")
let $datetime := dateTime(data($r))
where year-from-dateTime($datetime) ge 2003
and month-from-dateTime($datetime) eq 12
and day-from-dateTime($datetime) eq 25
return $r

```

Listing 8: Selection Query (Q0b)

Q0: In this query (Listing 8), the user asks for historical data from all the weather stations by selecting all December 25 weather measurement readings from 2003 on.

Q0b is a variation of *Q0* where the input path (1st line in Listing 8), is extended by a *value* expression ("date").

```
for $r in collection ("/sensors") ("root")()
("results")()
where $r("dataType") eq "TMIN"
group by $date:= $r("date")
return count($r("station"))
```

Listing 9: Aggregation Query (Q1)

```
for $r in collection ("/sensors") ("root")()
("results")()
where $r("dataType") eq "TMIN"
group by $date:= $r("date")
return count(for $i in $r return $i("station"))
```

Listing 10: Aggregation Query (Q1b)

Q1: This query (Listing 10), finds the number of stations that report the lowest temperature for each date. The grouping key is the date field of each object.

Q1b is a variation of *Q1* that has a different returned result structure.

Q2: This self-join query (Listing 11) joins two large collections, one that maintains the daily minimum temperature per station and one that contains the daily maximum temperature per station. The join is on the station id and date and finds the daily temperature difference per station and returns the average difference over all stations.

```
avg(
  for $r_min in collection("/sensors")("root")()
  ("results")()
  for $r_max in collection("/sensors")("root")()
  ("results")()
  where $r_min("station") eq $r_max("station")
  and $r_min("date") eq $r_max("date")
  and $r_min("dataType") eq "TMIN"
  and $r_max("dataType") eq "TMAX"
  return $r_max("value") - $r_min("value")
) div 10
```

Listing 11: Join-Aggregation Query (Q2)

5.3 Single Node Experiments

To explore the effects of the various rewrite rules we first consider a single node, one core environment (i.e. one partition) and progressively enable the different sets of rules. We start by considering just the *path expression rules*. Figure 13 shows the execution time for all five queries with and without these rules. For this experiment, we used a 400MB collection of sensor files (each of size 10MB). Note that for these experiments we used a relatively small collection size since without the JSONiq rules Hyracks would need to process the whole (possibly large) file thus slowing its performance. The application of the Path Expression Rules results to a clear improvement of the execution time for all queries. These rules decrease the buffer size between operators since large sequences of objects are avoided. Instead, each object is passed on to the next operator separately, resulting in faster query execution.

Using the same dataset and having enabled the Path Expression rules, we now examine the effect of adding the *Pipelining* rules (Figure 14). We observe that in all cases the pipelining rules provide a drastic improvement (note the logarithmic scale!),

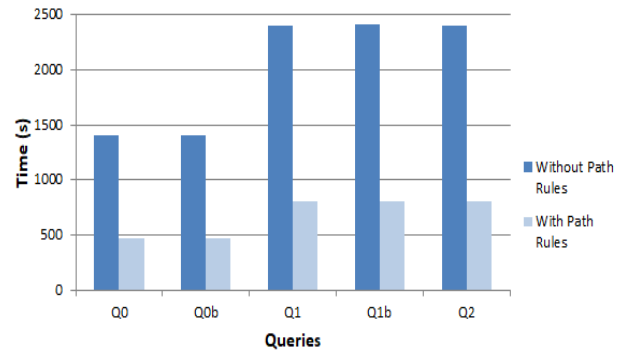


Figure 13: Execution Time before and after Path Expression Rules

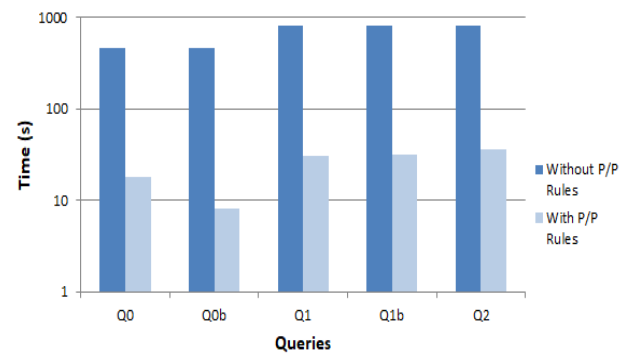


Figure 14: Execution Time (logscale) before and after the Pipelining Rules

speeding queries up by about two orders of magnitude. Applying these rules decreases the initial buffering requirement since we don't store the whole JSON document anymore, but just the matching objects. It is worth noting that the best performance is achieved by *Q0b*. *Q0b* stores in memory only the parts of the objects whose key field is "date". By focusing only on the "date", this gives the DATASCAN operator the opportunity to iterate over much smaller tuples than *Q0*. Clearly, the smaller the argument given to DATASCAN, the better for exploiting pipelining.

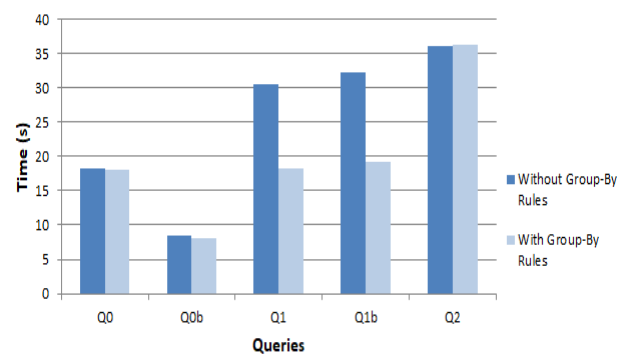


Figure 15: Execution Time before and after Group-by Rules

Having enabled both the path expression and the pipelining rules, we proceed considering the effect of adding the *Group-by* rules. The results are depicted in Figure 15. Clearly Q0, Q0b and Q2 are not affected since the Group-by rules do not apply. The Group-by rules improve the performance of both queries Q1 and Q1b. The improvement for both queries comes from the same rule, the rule that pushes the COUNT function inside the group-by. We note that the second Group-by rule, the one performing conversion, applies only to Q1; we do not enjoy an improvement from the conversion rule here because Q1b is already written in an optimized way. It is worth mentioning that the efficiency of the group-by rules depends on the cardinality of the groups created by the query. Clearly, the larger the groups, the better the observed improvement.

To study the effectiveness of all of the rewrite rules as the partition size increases, we ran an experiment where we varied the collection size from 100MB to 400MB. Figure 16 (again with a log scale) depicts the execution time for query Q1 both before and after applying all three sets of rewrite rules. We chose Q1 here because it is indeed affected by all of the rules. We can see that the system scales proportionally with the dataset size and that the application of the rewrite rules results in a huge query execution time improvement.

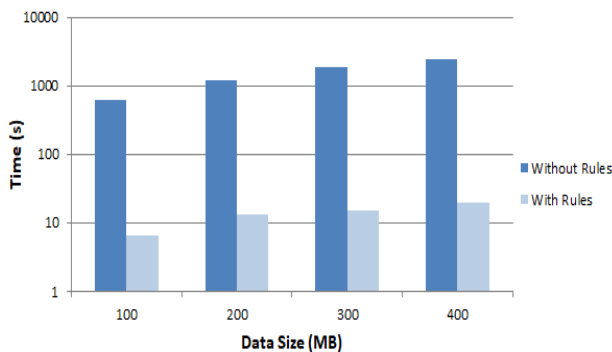


Figure 16: Execution Time (logscale) for Q1 before and after Rewrite Rules for different Data Sizes

From the above experiments, we can conclude that the Pipelining rules provide the most significant impact. It is also worth noting that the execution of the rewrite rules during query compilation adds a minimal overhead (just a few msec) to the overall query execution cost.

Single node Speed-up: To test the system's single node speed up, we used a dataset larger than our node's available memory space (8GB). Specifically, we used 88GB of JSON data, which we progressively divided from one up to eight partitions (our CPU has 4 cores and supports up to 8 hyperthreads). Each partition corresponds to a thread. The results are shown in Figure 17.

For up to 4 partitions and for almost all query categories, we achieve good speed-up since our observed execution time is reduced by a factor close to the number of threads that are being used. On the other hand, when using 8 hyperthreaded partitions we observe no performance improvements and in some cases a slightly worst execution time. This is because our processing is CPU bound (due to the JSON parsing), hence the two hyperthreads are effectively run in sequence (on a single core). In summary, we see the best results when we match the number of partitions to the number of cores.

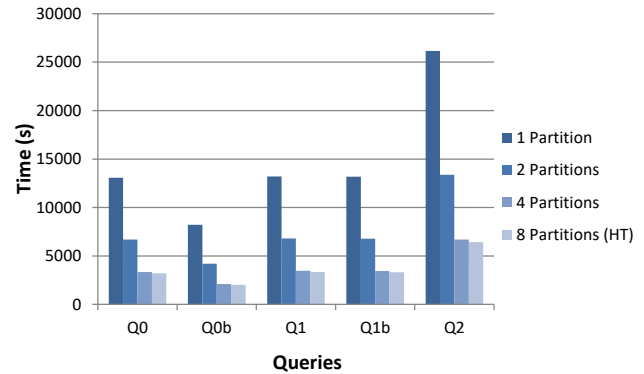


Figure 17: Single Node Speed-up

Comparison with MongoDB and AsterixDB: When comparing our performance against MongoDB and AsterixDB we observed that the performance of these systems is affected by the structure of the input JSON file. For example, a file with the structure of Listing 6 will be perceived by MongoDB and AsterixDB as a single, large document. Since MongoDB and AsterixDB are optimized to work with smaller documents (MongoDB has in addition a document size limitation of 16MB), to make a fair comparison we examined their performance while varying the number of documents per file.

We first unwrapped all the JSON items inside "root" (Listing 6). This results to many individual documents per file, each document containing a "metadata" JSON object and its corresponding "results" JSON array (typically with 30 measurements). We further manipulated the number of documents per file by varying the number of member objects (measurements) inside the "results" array from 30 (one month of measurements per document) to 1 (one day/measurement per document).

Figure 18.a depicts the query time performance for query Q0b for VXQuery, MongoDB, AsterixDB and AsterixDB(load); the space used by each approach appears in Figure 18.b. The total size of the dataset is 88GB and we vary the measurements per JSON array.

In contrast to MongoDB and the AsterixDB approaches, we note that the performance of VXQuery is independent of the number of documents per file. MongoDB has better query time for larger documents (30 measurements per array). Since MongoDB performs compression per document, larger documents allow for better compression and thus query time performance. This can also be seen in figure 18.b, where the space requirements increase as the document becomes smaller (and thus less compression is possible). The space for both AsterixDB approaches and VXQuery is independent from the document size (which is to be expected as currently these systems do not use compression).

AsterixDB shows a different query performance behavior than MongoDB. Its best performance is achieved for smaller document sizes (one measurement per document). Since it shares the same infrastructure as VXQuery, the difference in its performance relative to VXQuery is due to the lack of the JSONiq Pipeline Rules. Without them, the system waits to first gather all the measurements in the array before it moves them to the next stage of processing. This holds for both AsterixDB and AsterixDB(load). Among the two approaches, the AsterixDB(load) approach has better query performance since it is optimized to work better for data that is already in its own data model. Interestingly, for the

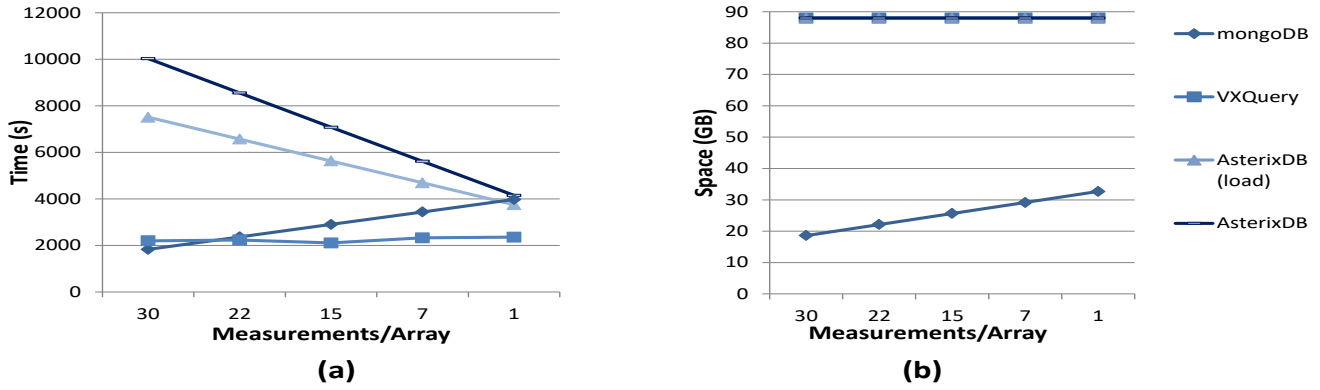


Figure 18: (a) Execution Time and (b) Space Consumption for Different Measurement Sizes per Array

Measurements/Array	30	22	15	7	1
MongoDB	9000	11703	14443	17146	19876
AsterixDB (load)	24659	23987	24205	24547	24612

Table 1: Loading Time in sec for AsterixDB (load) and MongoDB for Different Record Sizes

smaller document sizes (where compression is limited), AsterixDB and MongoDB have similar query performance. For larger document sizes their query performance difference seems to be directly related to their data sizes. For example, with 30 measurements per document, MongoDB uses about 4.5 times less space due to compression and has about 4 times less query time than AsterixDB(load).

Table 1 depicts the loading times for MongoDB and AsterixDB(load) for different measurements/array (in contrast there is no loading time for AsterixDB and VXQuery). The different loading times can also be explained by the space consumed by MongoDB and AsterixDB(load) (Figure 18.b). Specifically, MongoDB needs less loading time due to its use of compression; as expected, its loading time increases as the number of measurements per array is decreased due to less compression.

Comparison with SparkSQL: We next compare with a well-known NoSQL system, SparkSQL. In this experiment we ran Q1 both on VXQuery with the JSONiq rewrite rules and on Spark SQL and we compared their execution times. We used a single node and one core as the setup for both systems. We varied the dataset sizes starting from 400MB up to 1GB. We could not run experiments with larger input files because Spark required more than the available memory space to load such larger datasets.

Table 2 shows the SparkSQL loading times for the datasets used in this experiment. Figure 19 shows the query times for query Q1 for the different data sizes. Note that the VXQuery bar shows the total execution time for each file (which includes the loading and query processing) while the SparkSQL bar corresponds to the query processing time only. VXQuery’s total execution time is slower than Spark’s query time for small files. The two systems show similar performance for 800MB files. However, as the collection size increases, Spark’s behavior starts to deteriorate. For the 1GB dataset our system’s overall performance is clearly faster. If one counts also for the file loading time of SparkSQL (the overhead added by loading and converting JSON data to a SQL-like format), the VXQuery performance is faster. While the overhead of the loading phase is not a significant burden for

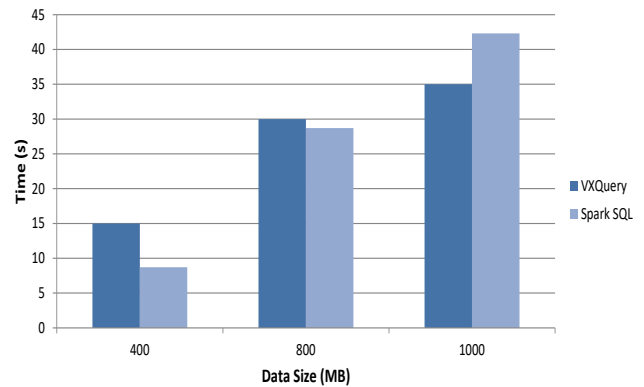


Figure 19: Spark SQL vs VXQuery Execution Time for Query Q1 Using Different Data Sizes (MB)

Data Size (MB)	Loading Time (s)
400	6.3
800	15
1000	40

Table 2: Loading Time for Spark SQL

Data Size (MB)	Spark Memory (MB)	VXQuery Memory (MB)
400	5650	1690
800	6230	1750
1000	7953	1760

Table 3: Data size to system memory in MBs

SparkSQL when considering small inputs (400MB) it becomes an important limiting factor even for medium size files (800MB).

We also examined the memory allocated by both systems (Table 3). The results show that VXQuery stores only data relevant to the query in memory, as opposed to SparkSQL, which stores everything. For file sizes above 2GB, the memory needs of SparkSQL exceeded the node’s available 16GB, so it was unable to load the input data so as to query it.

5.4 Cluster Experiments

The goal of this section is to examine the cluster speed-up and scale-up achieved by VXQuery due to our JSONiq rewrite rules

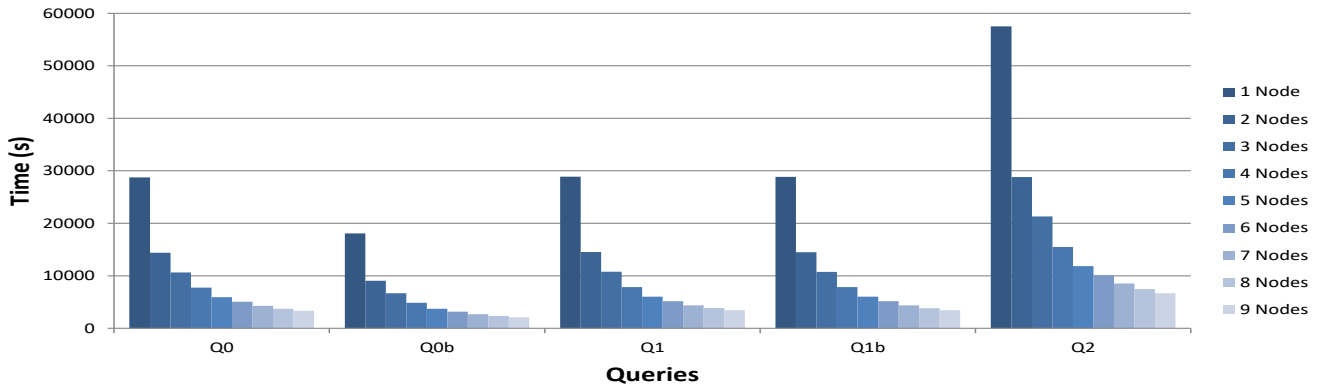


Figure 20: VXQuery Cluster Speed-up for all Queries (803GB Dataset)

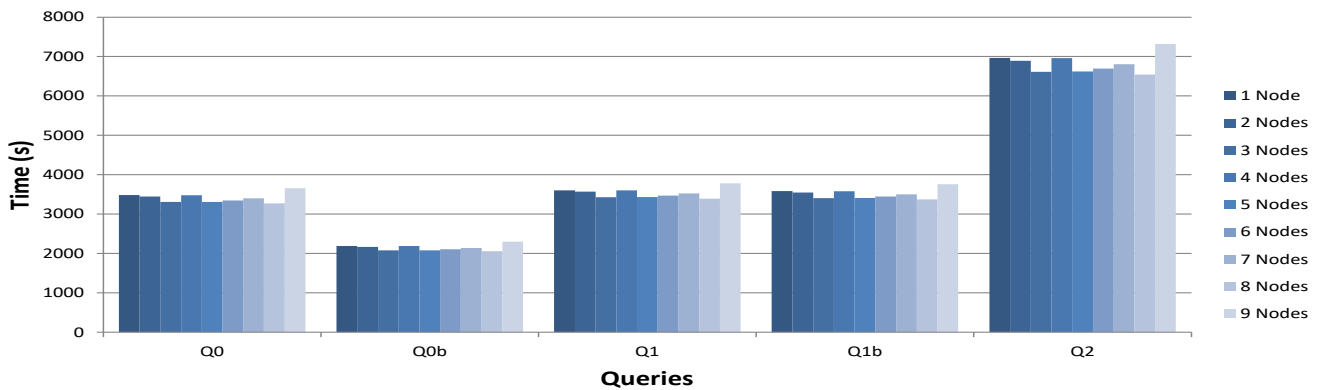


Figure 21: VXQuery Cluster Scale Up for all Queries (88GB per Node)

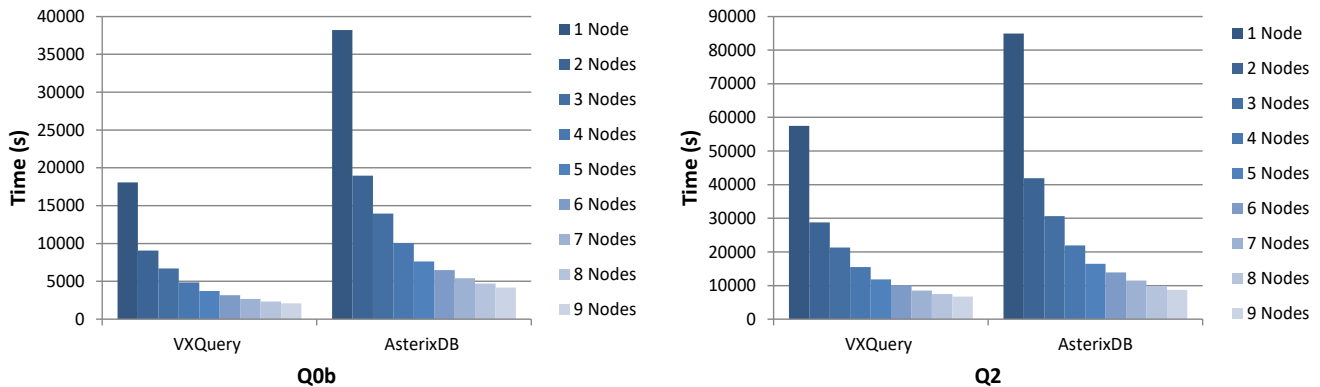


Figure 22: VXQuery vs AsterixDB: Cluster Speed-up for Q0b and Q2 (803GB Dataset)

and compare it with AsterixDB and MongoDB. For all the following experiments we used 4 partitions per node which achieves the best execution time as shown in the previous section.

To measure the *cluster speed-up* we started with a single node and extended our cluster by one node until it reached to 9 nodes. We used 803GB of JSON weather data which were evenly partitioned among the nodes used in each experiment. This dataset exceeds the available cluster memory. The results for this evaluation are shown in Figure 20. We note that in all the cases cluster speed-up is proportional to the number of nodes being used, without depending on the type of the query. We observe that the last query (Q2) takes the most time to execute. This is expected because Q2 is a self join query, which means that it has to process

twice the amount of data. On the other hand, for VXQuery, Q0b has the best response time due to its small input search path as described in previous sections.

To show the *scalability* achieved by VXQuery, we started with a dataset of size 88GB which exceeds one node's available memory (8GB). With each additional node added we add four partitions totaling 88GB (hence when using 9 nodes the whole collection is 803GB). The results appear in Figure 21. As it can be seen our system achieves very good scale-up performance; the query execution time remains roughly the same, which means that the additional data is processed in roughly the same amount of time.

Comparison with AsterixDB: In the cluster experiments, we compare against AsterixDB (i.e. without loading; each dataset

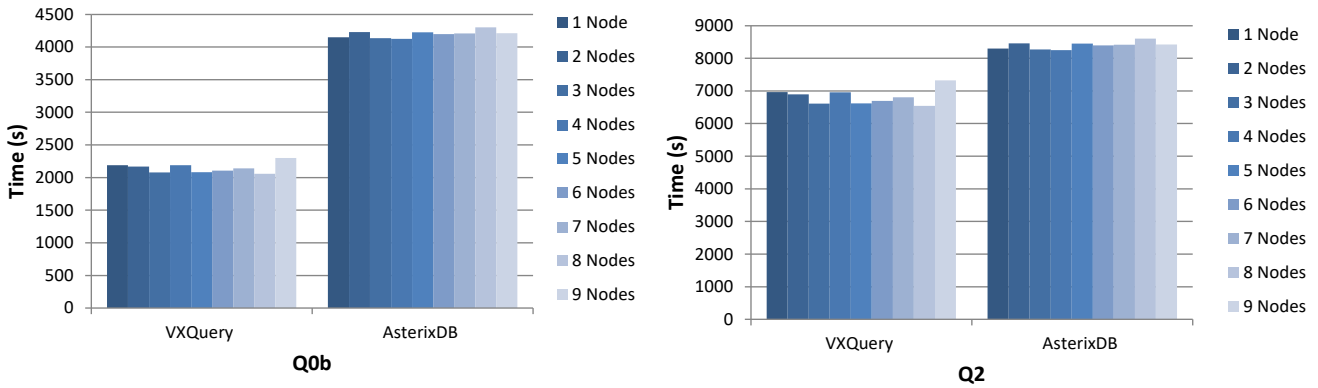


Figure 23: VXQuery vs AsterixDB: Cluster Scale-up for Q0b and Q2 (88GB per Node)

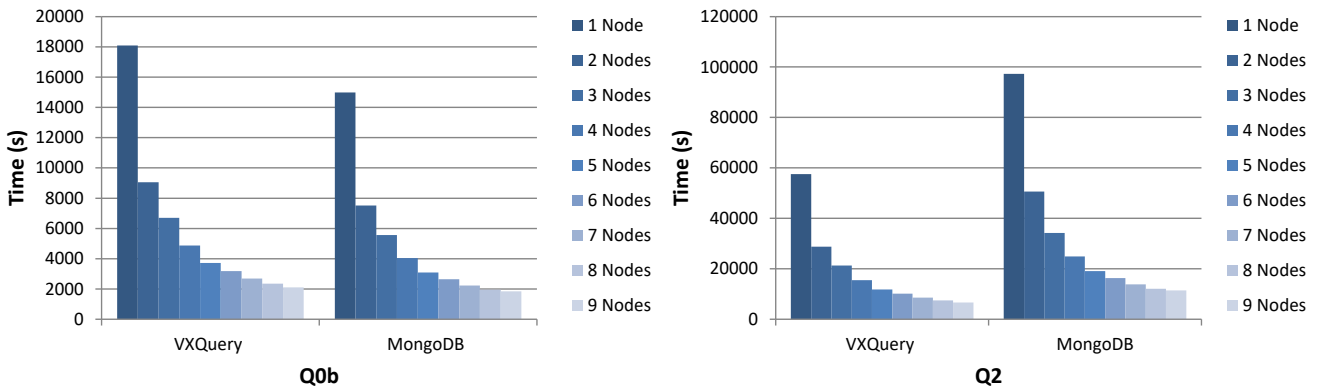


Figure 24: VXQuery vs MongoDB: Cluster Speed-up for Q0b and Q2 (803GB Dataset)

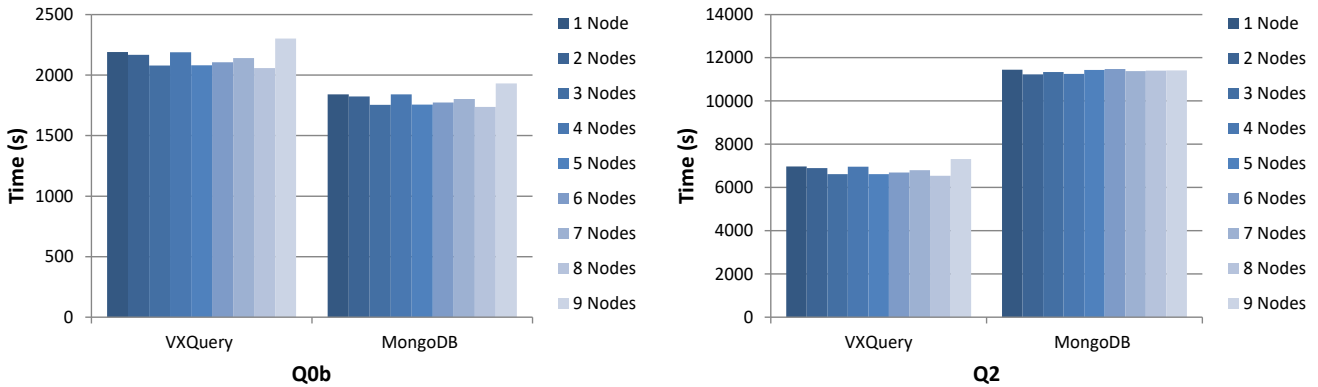


Figure 25: VXQuery vs MongoDB: Cluster Scale-up for Q0b and Q2 (88GB per Node)

is provided as an external data source). As seen in the single-node experiments, the best performance for AsterixDB is achieved when "results" consists of only one measurement; thus we use this structure for the following evaluation.

Following similar reasoning with the single-node comparison, we observe that VXQuery performs better both for speed up (Figure 22) and scale up (Figure 23), using queries Q0b and Q2 as representative examples.

Comparison with MongoDB: Similarly, we compare against the MongoDB configuration that achieved the best performance in the single-node experiments (i.e., "results" contains all monthly measurements). Overall, MongoDB has faster query time for selection queries than VXQuery (Figure 24 shows speedup for query

Q0b; the Q0 query performed similarly). Since MongoDB performs a compression during the loading phase of the dataset, the dataset stored in the database is much smaller giving an advantage to selection queries. However, VXQuery's execution time for query Q0b is still comparable since its small input search path gives the opportunity for the Pipeline rules to be exploited.

In contrast, VXQuery has a faster execution time than MongoDB on join queries (like Q2). For this self-join, MongoDB tries to put all the documents that share the same station and date in the same document; thus creating huge documents which exceed the 16MB document size limit causing it to fail. To overcome this problem, we perform an additional step before the actual join. We unwind the "results" array and we project only the

Data Size (GB)	Loading Time (sec)
88	9000
803	81000

Table 4: Loading Time for MongoDB

necessary fields. After that, we perform the actual join on the corresponding attributes (i.e. station, date of measurement).

On the other hand, in VXQuery there is no document size limitation, making VXQuery more efficient in handling large JSON items. Table 4 shows the MongoDB loading times per node. This adds a huge overhead to the performance of the overall system and it can be prohibitively large for real-time applications where the dataset may not be known in advance.

Comparison with SparkSQL: As mentioned in the single node experiments SparkSQL could not run experiments with larger input files because of the required memory space to load such larger datasets. Hence we omit multi-node experiments with SparkSQL, since the dataset size will be very small to indicate and difference in the execution time.

6 CONCLUSIONS AND FUTURE WORK

In this work we introduced two categories of rewrite rules (path expression and pipelining rules) based on the JSONiq extension to the XQuery specification. We also introduced a third rule category, group-by rules, that apply to both XML and JSON data. The rules enable new opportunities for parallelism by leveraging pipelining; they also reduce the amount of memory required as data is parsed from disk and passed up the pipeline. We integrated these rules into an existing system, the Apache VXQuery query engine. The resulting query engine is the first that can process queries in an efficient and scalable manner on both XML and JSON data. Through experimentation, we showed that these rules improve performance for various selection, join and aggregation queries. In particular, the pipelining rules improved performance by several orders of magnitude. The system was also shown both to speed-up and scale-up effectively. Moreover, when compared with other systems that can handle JSON data, VXQuery shows significant advantages. In particular, our system can directly process JSON data efficiently without the need to first load it and transform it to an internal data model.

We are currently working on supporting indexing over both types of data (XML and JSON). Indexing presents a significant challenge, as there is no simple way to decide the level at which an object could be indexed; indexing will further improve the system’s performance since the searched data volume will be significantly reduced. All of the code for our JSONiq extension is available through the Apache VXQuery site [4] and it will be included in the next Apache VXQuery release. Furthermore, we plan to add the proposed path and pipelining rules directly to AsterixDB given that it shares the same infrastructure (Algebricks and Hyracks) with VXQuery.

7 ACKNOWLEDGMENTS

This research was partially supported by NSF grants CNS-1305253, CNS-1305430, III-1447826 and III-1447720.

REFERENCES

- [1] 2012. Jackson project. <https://github.com/FasterXML/jackson>. (2012).
- [2] 2013. Zorba. <http://www.zorba.io/home>. (2013).
- [3] 2014. Apache Parquet. <https://parquet.apache.org/>. (2014).
- [4] 2016. Apache VXQuery. <http://vxquery.apache.org/>. (2016).
- [5] 2017. Apache AsterixDB. <https://asterixdb.apache.org/>. (2017).
- [6] 2017. Apache Drill. <https://drill.apache.org/>. (2017).
- [7] 2017. Apache Spark. <https://spark.apache.org/>. (2017).
- [8] 2017. JSONiq Extension to XQuery. <http://www.jsoniq.org/docs/JSONiqExtensionToXQuery/html-single/index.html>. (2017).
- [9] 2017. JSONiq Language. <http://www.jsoniq.org/docs/JSONiq/html-single/index.html>. (2017).
- [10] 2017. MongoDB. <https://www.mongodb.com/>. (2017).
- [11] 2017. Postgres-XL. <https://www.postgres-xl.org/>. (2017).
- [12] 2017. PostgreSQL. <https://www.postgresql.org/>. (2017).
- [13] Michael Armbrust and others. 2015. Spark SQL: Relational data processing in Spark. In *Proceedings of ACM SIGMOD Conference*. 1383–1394.
- [14] Kevin S Beyer, Vuk Ercegovic, Rainer Gemulla, Andrey Balmin, Mohamed Eltabakh, Carl-Christian Kanne, Fatma Ozcan, and Eugene J Shekita. 2011. Jaql: A scripting language for large scale semistructured data analysis. In *Proceedings of VLDB Conference*.
- [15] Vinayak Borkar, Yingyi Bu, E Preston Carman Jr, Nicola Onose, Till Westmann, Pouria Pirzadeh, Michael J Carey, and Vassilis J Tsotras. 2015. Algebricks: a data model-agnostic compiler backend for Big Data languages. In *Proceedings of 6th ACM Symposium on Cloud Computing*. 422–433.
- [16] Vinayak Borkar, Michael Carey, Raman Grover, Nicola Onose, and Rares Vernica. 2011. Hyracks: A flexible and extensible foundation for data-intensive computing. In *27th International Conference on Data Engineering*. 1151–1162.
- [17] E Preston Carman, Till Westmann, Vinayak R Borkar, Michael J Carey, and Vassilis J Tsotras. 2015. A scalable parallel XQuery processor. In *IEEE International Conference on Big Data*. 164–173.
- [18] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, Danyel Fisher, John C Platt, James F Terwilliger, and John Wernsing. 2014. Trill: A high-performance incremental query processor for diverse analytics. *Proceedings of the VLDB Endowment* 8, 4 (2014), 401–412.
- [19] Craig Chasseur, Yanan Li, and Jignesh M Patel. 2013. Enabling JSON Document Stores in Relational Systems.. In *WebDB*, Vol. 13. 14–15.
- [20] Gabriel Filios, Sotiris Nikolettas, Christina Pavlopoulou, Maria Rapti, and Sébastien Ziegler. 2015. Hierarchical algorithm for daily activity recognition via smartphone sensors. In *2nd IEEE World Forum on Internet of Things (WF-IoT)*. 381–386.
- [21] Felix N Kogan. 1995. Droughts of the late 1980s in the United States as derived from NOAA polar-orbiting satellite data. *Bulletin of the American Meteorological Society* 76, 5 (1995), 655–668.
- [22] Shamanth Kumar, Fred Morstatter, and Huan Liu. 2013. *Twitter Data Analytics*. Springer Publishing Company.
- [23] Yanan Li, Nikos R Katsipoulakis, Badrish Chandramouli, Jonathan Goldstein, and Donald Kossmann. 2017. Mison: a fast JSON parser for data analytics. In *Proceedings of the VLDB Endowment*, Vol. 10. 1118–1129.
- [24] Jimmy Lin and Dmitriy Ryabov. 2013. Scaling big data mining infrastructure: the twitter experience. *ACM SIGKDD Explorations Newsletter* 14, 2 (2013), 6–19.
- [25] Zhen Hua Liu, Beda Hammerschmidt, and Doug McMahon. 2014. JSON data management: supporting schema-less development in RDBMS. In *Proceedings of ACM SIGMOD Conference*. 1247–1258.
- [26] Jason McHugh and Jennifer Widom. 1999. Query Optimization for XML. *Proceedings of the 25th VLDB Conference* (1999).
- [27] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 330–339.
- [28] Nurzhan Nursetov, Michael Paulson, Randall Reynolds, and Clemente Izurieta. 2009. Comparison of JSON and XML data interchange formats: a case study. *22nd Intl. Conference on Computer Applications in Industry and Engineering (CAINE)* (2009), 157–162.
- [29] Daniel Tahara, Thaddeus Diamond, and Daniel J Abadi. 2014. Sinew: a SQL system for multi-structured data. In *Proceedings of ACM SIGMOD Conference*. ACM, 815–826.

An Automated System for Internet Pharmacy Verification

Alberto Cordioli

Prometeia

alberto.cordioli@prometeia.com

Themis Palpanas

Paris Descartes University

themis@mi.parisdescartes.fr

ABSTRACT

In the past years, we have witnessed an explosion of web applications, and in particular of electronic commerce websites. This has led to unquestionable benefits for both producers and consumers of goods. On the other hand, however, untrusted companies have the opportunity to bypass checks and regulations imposed by relevant bodies. This problem is prevalent in the context of online commerce of pharmaceutical products, where it is essential that such products are safe, of good quality, and only used with a proper prescription. In this work, we study the problem of internet pharmacy verification. To this effect, we build a classifier, able to find patterns and predict the class of unseen data. Moreover, we devise algorithms that give a trust score to each pharmacy, in order to have a legitimacy indicator usable by human reviewers. We experimentally evaluate the proposed approach with real data coming from two different time periods. The results demonstrate the effectiveness of our approach, as well as the potential of using similar techniques for automatically checking regulation compliance in electronic commerce.

1 INTRODUCTION

The growth of web-related technologies, and in particular e-commerce, has offered companies the opportunity to increase their own business, selling directly their products and goods to customers within and across borders. Even though this has led to unquestionable benefits for customers, untrusted companies can also access the market and sell products, for which it is not always possible to assess the quality.

The above problem is even more prominent when we are dealing with pharmaceutical products. Online sale of counterfeit drugs has become an important problem, with studies by the World Health Organization (WHO) showing that in more than 50% of the cases, drugs sold by websites that conceal their physical address are counterfeit¹. The WHO argues that counterfeiting occurs both with branded and with generic products, and counterfeit drugs may include the correct ingredients but fake packaging, the wrong ingredients, insufficient active ingredients, or no active ingredients whatsoever². Evidently, counterfeit drugs represent an enormous public health challenge [26], and also a major illicit economic activity, with an estimated \$75 billion market for 2010³.

Moreover, the mere task of distinguishing a *legitimate* from an *illegitimate* online pharmacy is rather challenging. This is true for domain experts, and is often times impossible to do for simple users, especially since *illegitimate* pharmacies and drugs are designed to look like *legitimate* ones (including the packaging

of drugs, and the drugs themselves, which are usually identical to the original ones).

Figure 1 shows the front webpage of two online pharmacies, only one of which is *legitimate*. Evidently, a simple observation of the two webpages is not enough to reveal which one⁴. Hence, there is a pressing need for assessing the quality of pharmaceutical products sold online, and a major step in this direction is to assess the trustiness of online pharmacies, which is the focus of this study.

There are different factors that make a pharmacy *illegitimate*. In U.S.A. (as well as in many other countries), an online pharmacy must satisfy regulations and meet strict requirements. The requirements that are most frequently violated in the U.S. are, for example, the selling of products without prescriptions and the selling of drugs that are not “FDA-Approved”⁵ [21]. Evidently, checking these factors is not an easy task, especially for people that do not have any kind of competence and knowledge in this field, such as the normal consumers.

It is for this reason that specialized companies have made the verification of health-related websites their own business. LegitScript⁶, for example, offers an internet pharmacy verification service and collaborates with the major search engines (e.g., Google, Bing) in order to enforce policies against *illegitimate* online pharmacies, which can be as much as 90% of the total number of online pharmacies [21].

The process of classifying health-related websites into *legitimate* and *illegitimate* pharmacies is currently mostly *manual*, and requires a great investment of time and human resources. The increasingly large number of online pharmacies, and correspondingly large number of *illegitimate* online pharmacies, leads to the necessity of streamlining the review process with a system capable of automatically giving a trust score to online pharmacies. In this manner the system can assist the human reviewers, relieving them of some tedious and time-consuming tasks.

In this paper, we propose the first systematic approach to the aforementioned problem, using techniques that are based on both text and network features, and we describe a system capable of verifying internet pharmacies. We have made all our code publicly available⁷. Even though previous studies have discussed this problem (e.g., [3, 23, 28]), they did not provide algorithmic solutions for it.

The contributions we make in this paper are as follows.

- We provide the first systematic study for the problem of internet pharmacy verification and formalize two sub-problems: (a) classification of online pharmacies into *legitimate* and *illegitimate*; and (b) ranking online pharmacies according to a *legitimacy* score.
- We study and evaluate indicators that can distinguish between *legitimate* and *illegitimate* pharmacies. We propose

¹<http://who.int/bulletin/volumes/88/4/10-020410/en/>

²http://apps.who.int/iris/bitstream/10665/65892/1/WHO_EDM_QSM_99.1.pdf

³The complete article can be found at: <http://www.usatoday.com/money/industries/health/drugs/story/2011-10-09/cnbc-drugs/50690880/1>


© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

⁴The pharmacy depicted in Figure 1a is *illegitimate*, while the one depicted in Figure 1b is *legitimate*.

⁵If a drug is FDA-Approved it means that specific tests have been conducted to prove the quality of the product.

⁶<http://www.legitscript.com>




⁷<https://sites.google.com/view/acolfplg/home>

Q Type keyword here **Search**  Your cart: \$ 0.00 (0 items)

Search by letter: [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

[Bestsellers](#) [Testimonials](#) [FAQ](#) [Contact us](#) [Track Order](#)


Categories list **Today Bestsellers**

<p>» Alcoholism</p> <p>» Alzheimer's And Parkinson's</p> <p>» Analgesics</p> <p>» Anti-inflammatories</p> <p>» Antiallergic</p> <p>» Antibiotics</p> <p>» Anticonvulsants</p> <p>» Antidepressants</p> <p>» Antifungals</p>	<p>Kamagra® Sildenafil Citrate 100mg</p> <p></p> <p>Kamagra® is a new medicine manufactured by Ajanta Pharma (India) used for treating erectile disorders in men.</p> <p>\$ 1.50 Buy now!</p>	<p>Generic Betagan Levobunolol 0.5% 5ml</p> <p></p> <p>Generic Betagan is used for lowering eye pressure and treating glaucoma.</p> <p>\$ 10.00 Buy now!</p>	<p>Generic Ocuflax Ofloxacin 0.3% 5ml</p> <p></p> <p>Generic Ocuflax is used for treating and preventing eye infections associated with conjunctivitis (pink eye) and corneal ulcers caused by certain bacteria in patients 1 year</p> <p>\$ 5.00 Buy now!</p>
---	--	---	---

(a) front webpage of online pharmacy 1

drugstore.com BEAUTY.COM welcome: sign in your account your list™ | your Rx | help 2 sites 1 bag 0 items \$0.00 **checkout**

NEW! **FREE SHIPPING** on orders over \$25 – EVERY DAY! »


drugstore.com[®] the uncommon drugstore search keyword  [pharmacy](#) [photo](#) [contact lenses](#)


household & food medicine & health personal care beauty baby & mom vitamins diet & fitness sexual well-being fsa GNC green & natural sales & coupons

FREE shipping* **INSTANT coupons** **HUGE selection** **GET 5% back** **60,000+ products**

uncommon selection


With more than 60,000 products, you'll always find just what you're looking for. From the tried-and-true to the unique and unexpected, we've got you covered!

shop now 



SHOP BY category **SHOP BY brand** **SHOP BY top rated**

- staff picks
- customer favorites
- gift center
- your list™
- free gifts
- instant coupons
- request a product
- video library

FASTER shipping for military with thanks 

Boots The UK's #1 cosmetic & skin care brand »

(b) front webpage of online pharmacy 2

Figure 1: Examples of two online pharmacies.

novel features that are based on both the text of the online pharmacy website, and its network structure, and integrate these features in our models.

- We describe effective and efficient solutions for the classification and ranking problems. The proposed solutions can automate to a large extent the process of online pharmacy verification.
- We experimentally validate our methods with the two largest real datasets used in the literature, comprising of almost 2500 *illegitimate* pharmacies and 200 *legitimate* pharmacies, crawled in two different time periods. The results demonstrate the accuracy of our approach and its practical value, and showcase the potential of similar techniques in relevant problems in e-commerce.

The rest of this paper is organized as follows. In Section 2, we elaborate on existing work. In Section 3, we provide some background material that is necessary for the discussion that follows, and we formally define the two problems we solve. We describe our proposed solution in Sections 4 and 5. We present the experimental results in Section 6, and finally, we conclude in Section 7 with a brief summary and some thoughts on future work.

2 RELATED WORK

2.1 Pharmacy Verification

Previous works have discussed the problem of online pharmacy verification [22], and advocated the need for studying this problem from the regulation and public-health points of view [27, 28]. A study in the area of medicinal drug commerce has shown that consumers should be able to reduce their risk by relying on trusted lists compiled by credited agencies [3], and by using common sense when examining packaging and pills, while other studies have explored the problem of identifying controversial drugs, by monitoring consumer opinion [33, 34].

Nevertheless, the major problem is that it is not possible to assess the quality of a drug sold online, at least until it has been purchased. And even at that time, determining if such medicine is safe or not, requires several analyses and competences, often not held by the normal consumer. The most promising solution is the one of using official lists of trusted online pharmacies. Nevertheless, this is a daunting task to complete manually, because of the sheer number of online pharmacies and the rate with which they appear (or disappear).

Some studies have focused on the problem of identifying features and signals that distinguish *legitimate* and *illegitimate* online pharmacies [23, 27, 28]. In [23] the authors performed a comparative analysis of website trust features applied to the case of online pharmacies, which showed that proven *legitimate* pharmacies use more extensively verification seals and have more instances of health content than *illegitimate* pharmacies. On the other hand, *illegitimate* pharmacies have fewer store presence features than *legitimate* pharmacies. In [22] the authors try to understand the reasons for the success of unlicensed online pharmacies. They discover that instead of directly competing with licensed pharmacies, unlicensed pharmacies often sell drugs that licensed pharmacies do not, or cannot sell.

Additional studies [27, 28] demonstrate that the problem of online pharmacy verification should be studied at two different levels: from the regulation point of view and from the public health point of view. The very same conclusion has been outlined another study as well [27], where the authors perform a review of the scientific literature and a study of several scientific and institutional databases. They showed that the phenomenon is continuing to spread, and in order to enhance the benefits and minimize the risks, a 2-level approach could be adopted: first, from a policy point of view implementing international level laws, and second on an individual consumer point of view. Note that none of studies mentioned above propose algorithmic solutions for tackling the problem at hand.

CADRE [36] is a cloud-assisted drug recommendation system, which can recommend users with related drugs, according to their symptoms. The system clusters drugs into several groups according to the functional description information, and designs a basic personalized drug recommendation based on user collaborative filtering. While this system can help consumers find alternative drugs for their symptoms, it can not be used for identifying *illegitimate* drugs, or *illegitimate* online pharmacies.

In this study, we show that another direction is possible and effective. We claim that relevant bodies (e.g., LEAs and private agencies in health-care system) could use state-of-the-art data mining techniques in order to find, isolate, and eventually shut-down *illegitimate* online pharmacies. Our approach outlines a text classification process and a network trust algorithm in order to assess the *legitimacy* of internet pharmacies.

2.2 Text Classification and Network Trust Algorithms

Our problem is reminiscent to spam detection [5]. Even though the problems of internet pharmacy verification and web spam detection exhibit some similarities, they are very different in the definition of what is considered *legitimate*. In our case, we might require very specific knowledge in order to differentiate between *legitimate* and *illegitimate* examples. Moreover, the final scope of the two systems are different. In [5] the authors aim to improve search engines algorithms, while in this work the final users are domain-specific analysts working in the field of online pharmacy verification.

Text Classification (TC) is defined as the process, in which a document is automatically classified in one or more categories (classes) [1, 31]. In this process, a set of labeled data is used to train a classifier, which recognize patterns among instances of the same class. These patterns are then used to build a model able to classify unlabeled instances. TC have been used in many contexts, including language identification [6], message filtering (i.e., spam filtering) [2, 19, 25], hierarchical categorization of web pages [7, 11], and others. Recent studies have surveyed text classification algorithms [1], and also studied the behavior of classifiers in the presence of label noise [14, 24].

In the specific context of web content, the classification of web content in one or more classes has been studied by many researchers. In [17], the author analyzes the nature of web content and metadata in relation to requirements for text features, and presents a system for automatically classifying websites into industrial categories. The work presented in [11] explores the use of a hierarchical structure for classifying a large, heterogeneous collection of web content.

In [13], the authors compare three different text representation techniques, based on (character) N-Gram Graphs, the Term Vector model, the Character N-Grams model, and the N-Gram Graphs model, with respect to three different categories of documents: curated, semi-curated and raw documents. They show that each category calls for different classification settings with respect to the representation model; moreover they show that N-Gram Graphs model achieves higher performances on each of the three different categories analyzed. This is a versatile technique that we use in our work, and that we further discuss in the following sections.

Assessing the trustiness in a network of hosts or websites has become very important in the web context, where the number of web spam pages increases by the minute. In order to address this problem, some search engines have adopted trust algorithms to reduce the rank of such pages in query results. TrustRank [15] is a link analysis technique for semi-automatically separating useful webpages from spam. Starting from a seed of reputable web pages, TrustRank uses the underlying network structure to discover other pages that are likely to be legitimate.

In [20] the authors provide a variation of TrustRank algorithm, called Anti-TrustRank, where non-reputable web pages are selected as initial seed. A different algorithm, which is able to decrease the number of downloads of inauthentic files in a peer-to-peer file-sharing network, is presented in [18].

An important characteristic of our domain is that the two classes - *legitimate* and *illegitimate*- are strongly imbalanced: the number of *legitimate* examples represents only a small percentage of the total. The effect of skewed distribution has been studied in many aspects. In [35] the authors analyzed the effect

of class distribution on classifier learning and showed that the naturally occurring class distribution is often not the best choice for learning. In [10, 29, 32] the effects of dataset distribution have been studied for two very well known classifiers: C4.5 and SVM. In our work, we compare the results obtained by training classifiers with the natural distribution and with resampling (both undersampling and oversampling).

3 PRELIMINARIES AND PROBLEM DEFINITION

In this section, we recap some concepts and techniques needed for the rest of the paper, and formally define the problems we solve.

3.1 Preliminaries

We first define the concept of an online pharmacy, and we point out the differences between *legitimate* and *illegitimate* online pharmacies. This is important, since the term *illegitimacy* may have different interpretations depending on the contexts.

An online pharmacy is a website that offers medical products for sale. In this context there are different levels of *illegitimacy* for a pharmacy, and such levels depend on certain features that, if present, contribute to make that pharmacy *illegitimate*. We can group *illegitimate* pharmacies in three big categories:

- online pharmacies that do not adhere to accepted standards of medicine and/or pharmacy practice, including standards of safety;
- online pharmacies that violate, appear to violate, encourage violation of, or are not in compliance with applicable national or regional laws or regulations;
- online pharmacies that engage in fraudulent or deceptive business practices.

The first two categories are self explanatory. They include pharmacies that represent a threat for the people’s health, as they sell drugs that are not approved, or they are not in compliance with national or regional regulations. The third category is a more general class that includes those websites that scam people, stealing their data, or money (obviously, these websites are not only health-related). These three categories are not mutually exclusive, and a *illegitimate* pharmacy may belong to more than one, which is actually true for the majority of them.

Signals that make a pharmacy more likely to be *illegitimate* include concealing its physical address, being isolated from major trusted websites, as well having fewer store presence features, and fewer health-related text content than *legitimate* pharmacies [3, 28].

3.2 Problem Statement

We now formalize the problems that we solve in this study: first, classification of online pharmacies in *illegitimate* and *legitimate*; and second, ranking of online pharmacies according to their *legitimacy*.

We denote with \mathcal{P} the set of all the pharmacy websites. Let’s call \mathcal{P}^+ and \mathcal{P}^- the *legitimate* and the *illegitimate* pharmacies, respectively, in \mathcal{P} . We also suppose to know the class of a subset of pharmacies $\mathcal{P}_0 \subseteq \mathcal{P}$, i.e., there exists an *oracle function* O that for all the $p \in \mathcal{P}_0$:

$$O(p) = \begin{cases} 1 & \text{if } p \in \mathcal{P}^+ \\ 0 & \text{if } p \in \mathcal{P}^- \end{cases} \quad (1)$$

Assuming that a human reviewer can evaluate if a website is *legitimate* or not, we can think of the oracle function O as an evaluation performed by a human reviewer. However, oracle invocations are expensive and we cannot call O for each pharmacy in \mathcal{P} , so we aim to find another function, T , such that for some model Π :

$$T(p) = \begin{cases} 1 & \text{if } \Pi \models p \in \mathcal{P}^+ \\ 0 & \text{if } \Pi \models p \in \mathcal{P}^- \end{cases} \quad (2)$$

We recall that the formal notion $\Pi \models p \in \mathcal{P}^+$ means that Π entails $p \in \mathcal{P}^+$, where Π is a model derived and built from the training set \mathcal{P}_0 with some learning algorithm.

Note that there exist infinite T derived by infinite Π , but we are interested in the one that maximizes some evaluation measure (e.g., number of correctly classified instances, recall, precision). We can now formalize the classification problem:

PROBLEM 1 (ONLINE PHARMACY CLASSIFICATION (OPC)). *Given a set of pharmacies \mathcal{P} divided in two classes \mathcal{P}^+ and \mathcal{P}^- , a set of “known” pharmacies $\mathcal{P}_0 \subseteq \mathcal{P}$, and an evaluation measure φ , we seek a model Π_i such that $\forall p \in \mathcal{P}$, $T_i(p)$ maximizes φ .*

The second problem we are trying to solve is a ranking problem. We want to give a trust score to each pharmacy, which we could then use to produce an ordered list. More formally, we want to define a *totally ordered set*, where for each pair of pharmacies $p_1, p_2 \in \mathcal{P}$ it holds that $score(p_1) \leq score(p_2)$ or $score(p_2) \leq score(p_1)$. The score for each pharmacy is computed by combining different models.

PROBLEM 2 (ONLINE PHARMACY RANKING (OPR)). *Given a set of pharmacies \mathcal{P} divided in two classes \mathcal{P}^+ and \mathcal{P}^- , a set of “known” pharmacies $\mathcal{P}_0 \subseteq \mathcal{P}$, and a list of models Π_1, \dots, Π_k , we seek a totally ordered set $\langle \mathcal{P}, \leq \rangle$ such that, for each pair of elements $p_1, p_2 \in \mathcal{P}$, if $score(p_1) \leq score(p_2)$, then p_1 is “less legitimate” than p_2 .*

We expect that the ordered list naturally divides the pharmacies \mathcal{P} in two subsets (i.e., *legitimate* and *illegitimate* pharmacies), with all the elements of one subset at the top of this list, and all the elements of the other subset at the bottom. Without loss of generality, in the following we will focus on a *legitimacy* relation, which builds a list with *legitimate* examples at the top and *illegitimate* ones at the bottom.

In the following sections, we discuss how we solve the two problems formalized above, namely the classification and the ranking problem.

4 ONLINE PHARMACY CLASSIFICATION

Our classification algorithm is based on features that are relevant to both the text contained in the website of the online pharmacy and the web network structure around it.

4.1 Text Analysis

In order to reduce the dimensionality of the problem, it is common practice in Text Classification (TC) to use *preprocessing* and *summarization*.

Preprocessing: In the preprocessing step we remove the stop words found in the documents. In this way the most common words, which could adversely affect classification accuracy, are removed. To do so we rely on Apache Lucene⁸ version 3.4.0. We also decided to not use stemming, as the text contains a lot

⁸<http://lucene.apache.org/>

of technical words and trademarks, and this technique causes undesirable side-effects.

Summarization: The process of transforming a set of web pages into a unique summary is called *summarization*. For each pharmacy, we merge the text content of all the pages crawled into a single document. This step produces documents that include a large number of terms; documents comprising of 160,000 terms are not unusual. In our experiments, we evaluate the performance of the classifier models when we use the entire content of the document (all terms), as well as subsets of it. During this phase, we generate subsamples of the summary document considering a limited number of terms by randomly selecting 100, 250, 1000 and 2000 terms.

We are now ready to describe the core steps of our classification approach. Considering pharmacy websites as documents, and the two classes, *legitimate* and *illegitimate*, as mutually exclusive, we map our problem into the TC problem. We first convert each document (i.e., the text contained in an online pharmacy website) in a format suitable for the classification phase. We study two different such models, the Term Vector model [30], and the N-Gram Graphs model [12], both outlined below. Then, we use TC to classify unseen data relying on models built using the subset of labeled data \mathcal{P}_0 . We employ different learning algorithms to build a two-class classifier, including Naïve Bayesian Multinomial (NBM), Support Vector Machine (SVM), and the C4.5 decision tree learning algorithm.

4.1.1 Term Vector Model :

The Term Vector model is the most widely used text representation model in information retrieval, due to its high level of performance and scalability [30]. The model works as follows: given a set of documents \mathcal{D} , each document $d \in \mathcal{D}$ is represented as a vector of words $v_d = (v_1, v_2, \dots, v_{|\mathcal{W}|})$, where \mathcal{W} is the set of all the distinct words in \mathcal{D} . Each position v_i with $1 < i < |\mathcal{W}|$ in the vector represents the presence, or the absence, of word w_i in document d .

There are many variants to represent d_i values, but the most popular is the TF-IDF approach, which takes into account the number of occurrences of a term in a document (term frequency), and its overall frequencies in the whole set of documents \mathcal{D} (inverse document frequency).

4.1.2 N-Gram Graphs :

The N-Gram Graph is a graph $\mathcal{G}(V, E)$ which has character n-grams as its vertices, while the edges connecting the n-grams indicate proximity of the corresponding vertex N-Grams [12]. The weights on the edges represent how much the two N-Gram are related (one way of setting these weights is by counting how many times two N-Grams co-occur within a sliding window in the text). The advantage of N-Gram Graphs is that they conserve the order of the characters' appearance in the original text, and hence are more stable than the standard Character N-Gram Model. The three measures characterizing an N-Gram Graph are: (i) the minimum n-gram rank L_{min} , (ii) the maximum n-gram rank L_{max} , and (iii) the minimum neighborhood distance D_{win} . In our experiments, we use $L_{min} = L_{max} = D_{win} = 4$ [13].

In order to use N-Gram Graphs, we first transform each document d in a N-Gram Graph (refer to Figure 2). For each class c , we build an N-Gram Graph derived from merging the individual graphs in c , and we compute the similarities between each document d and the class graph. We then use the following similarity measures, namely, Containment Similarity (CS), Size Similarity

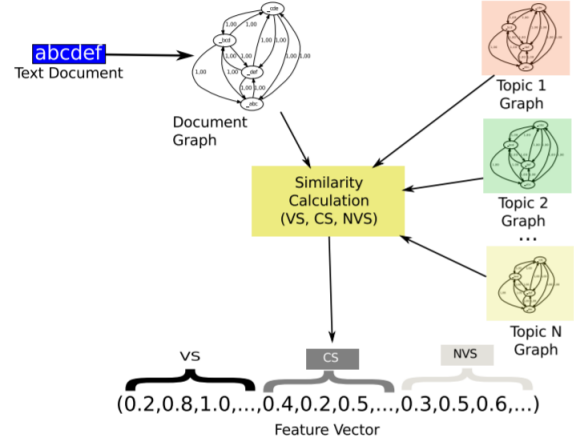


Figure 2: An overview of the classification process using N-Gram Graphs [13].

(SS), and Value Similarity (VS) in order to build a classifier able to predict the class of unseen data.

Containment Similarity (CS) expresses the proportion of edges of a graph G_i that are shared with a graph G_j .

$$CS(G_i, G_j) = \frac{\sum_{e \in G_i} \mu(e, G_j)}{\min(|G_i|, |G_j|)}$$

where e is an edge, and $\mu(e, G_i) = 1$ if, and only if $e \in G_i$. The cardinality $|G_i|$ here is intended as the number of edges of the graph G_i .

The ratio of sizes between two graphs is measured by the Size Similarity (SS):

$$SS(G_i, G_j) = \frac{\min(|G_i|, |G_j|)}{\max(|G_i|, |G_j|)}$$

We recall that with $|G_i|$ we indicate the number of edges in graph G_i

Value Similarity (VS) represents how many of the edges contained in G_i are contained in G_j , considering also the weight of such edges.

$$VS(G_i, G_j) = \frac{\sum_{e \in G_i} \frac{\min(w_e^i, w_e^j)}{\max(w_e^i, w_e^j)}}{\max(|G_i|, |G_j|)}$$

where w_e^i is the weight of edge e in the graph G_i .

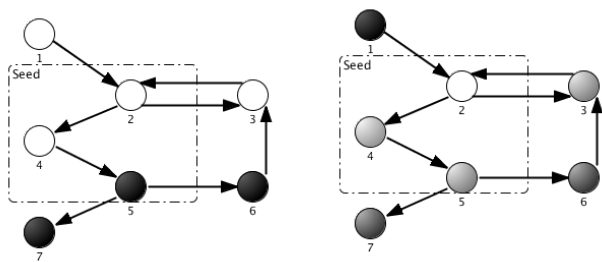
A combination of VS and SS gives another useful measure, called Normalized Value Similarity (NVS):

$$NVS(G_i, G_j) = \frac{VS(G_i, G_j)}{SS(G_i, G_j)}$$

4.2 Network Analysis

Apart from the text features described above that we use for the classification of online pharmacies, we additionally use features derived from the web network in which they are embedded. More specifically, we are interested in the links an online pharmacy web page has with other web pages: the web pages that this pharmacy points to, and the web pages that point to this pharmacy.

To this effect, we use features extracted from the TrustRank algorithm [15]. In TrustRank, the network is represented as a graph $\mathcal{G}(V, E)$, where the set of nodes V are websites (or more generally web pages) and the links between pages, represented by the set E , are drawn as directed edges. The algorithm computes a



(a) Initial state of a network made of "good" (white) and "bad" (black) nodes. (b) The network after the execution of the TrustRank algorithm, with different levels of "trustiness".

Figure 3: Illustration of the TrustRank algorithm.

Algorithm 1 Creates the network graph $\mathcal{G}(V, E)$

GRAPH-CREATION(\mathcal{P} : set of pharmacies)

```

1:  $V \leftarrow \emptyset$ 
2:  $E \leftarrow \emptyset$ 
3: for all Pharmacy  $p \in \mathcal{P}$  do
4:    $V \leftarrow V \cup p$ 
5:    $\mathcal{L} \leftarrow \text{outboundLinks}(p)$ 
6:   for all Link  $u \in \mathcal{L}$  do
7:      $V \leftarrow V \cup \text{endpoint}(u)$ 
8:      $E \leftarrow E \cup \{u\}$ 
9:   end for
10: end for

```

trust score for each node in the graph, based on the premise that "good" pages rarely point to "bad" ones (this property is also called *approximate isolation of good pages*). More specifically, TrustRank starts by selecting a seed of "known" pages, and gives a trust score of 1 to good pages and 0 to all others. After normalizing the values, the trust is propagated at each step until convergence. This process is illustrated in Figures 3a and 3b.

In order to run our version of TrustRank, we first need to construct the graph (refer to Algorithm 1), on which the algorithm will be applied.

Recall that the set \mathcal{P} contains both labeled and unlabeled examples. The *outboundLinks()* function (line 5) accepts the URL of an online pharmacy as input, and returns all the outbound links for this website, namely, all links that point to external domains⁹. The *endpoint()* function (line 7), returns the final destination of a link, extracting the second level domain.

For example, assume that *outboundLinks(p)* for some website p returns the following set of links: "http://www.medicalnewstoday.com/articles/238663.php", and "http://www.fda.gov/forconsumers/consumerupdates/ucm149202.htm". Then, the function *endpoint()* applied on each one of these URLs will return: "medicalnewstoday.com", and "fda.gov", respectively.

This step is important because it allows us to significantly prune the feature space, which in this case is represented by URLs. Please also note that this step will not affect the quality as we can assume all the pages belonging to the same domain having the same trustiness.

In a second step we have to assign an initial trust score to each node in the graph. In our graph we have 4 types of nodes:

- (1) *legitimate* nodes in \mathcal{P}_0 ; we denote this set with \mathcal{P}_0^+

⁹In other contexts outbound links are sometimes called *external links*, or *outcoming links*.

- (2) *illegitimate* nodes in \mathcal{P}_0 ; we denote this set with \mathcal{P}_0^-
- (3) unknown pharmacy nodes in $\mathcal{P} \setminus \mathcal{P}_0$
- (4) non-pharmacy nodes pointed to by nodes in \mathcal{P}

Note that the last category includes all the websites extracted from pharmacy links with functions *outboundLinks()* and *endpoint()*. Following our previous example, these are "medicalnewstoday.com" and "fda.gov".

During the initialization phase of TrustRank, we assign to all nodes in \mathcal{P}_0 the value returned by the oracle function, O , when invoked on these nodes. Hence, after the initialization, the known *legitimate* nodes (the first category of the list) have a trust score of 1, while all the other nodes have a value of 0. Finally, we can run TrustRank and train a classifier using the output values of corresponding nodes in \mathcal{P}_0 .

5 ONLINE PHARMACY RANKING

Our goal here is, given a pharmacy, to calculate a value that indicates the degree by which this pharmacy is *legitimate* or *illegitimate*. Having these values for all pharmacies will allow us to compute the *totally ordered set* sought in Problem 2. We assume that the list is ranked in decreasing order of *legitimacy* (if $p_1 \leq p_2$ then p_1 is "less *legitimate*" than p_2). We propose a cumulative model that combines the models built with text and network:

$$\text{rank}(p) = \text{textRank}(p) + \text{networkRank}(p).$$

When we use the Term Vector model with TF-IDF to represent documents, the *textRank* of a pharmacy p is computed as the membership probability of this instance p to the *legitimate* class, as estimated by a classifier solving Problem 1. For example, in the case of the Naïve Bayesian Multinomial classifier, the probability of a document d being in class c is computed as:

$$P(c | d) \propto P(c) \prod_{1 \leq k \leq n_d} P(t_k | c),$$

where $P(c)$ is the prior probability of class c , and $P(t_k | c)$ is the conditional probability of term t_k occurring in a document of class c . If the classifier is non-probabilistic, like for example SVM, we give to *textRank* a value of 1 if the instance is classified as *legitimate* and 0 if it is classified as *illegitimate*, which is the same as the output of function T .

On the other hand, when we use the N-Gram Graphs representation model, we compute *textRank()* using a different formula: rather than considering the output of the classifier, we sum up the graph similarity measures according to this formula:

$$\begin{aligned} \text{textRank}(p) = & CS_{\text{legitimate}}(p) + (1 - CS_{\text{illegitimate}}(p)) \\ & + SS_{\text{legitimate}}(p) + (1 - SS_{\text{illegitimate}}(p)) \\ & + VS_{\text{legitimate}}(p) + (1 - VS_{\text{illegitimate}}(p)) \\ & + NVS_{\text{legitimate}}(p) + (1 - NVS_{\text{illegitimate}}(p)) \quad (3) \end{aligned}$$

The abbreviations CS, SS, VS and NVS denote containment, size, value and normalized value similarities for the class in subscript, described in Section 4.1.

The function *networkRank()* simply returns the TrustRank value computed with the algorithms presented in Section 4.2.

6 EXPERIMENTAL EVALUATION

We now evaluate the proposed approach using real data provided by an American company, who is a leader in internet pharmacy verification. We will call this company *PharmaVerComp*. The pharmacies used in our study have been manually labeled as

	Dataset 1 Date 1	Dataset 2 Date 2 (6 months later)
# Examples	1459 (100%)	1442 (100%)
# Legitimate Examples	167 (12%)	167 (12%)
# Illegitimate Examples	1292 (88%)	1275 (88%)

Table 1: Datasets

legitimate or *illegitimate* by personnel of this company. Therefore, the dataset is consistent and error free.

All our code is publicly available at: <https://sites.google.com/view/acolfplg/home>.

6.1 Experimental Setup

At the time of this study, PharmaVerComp monitored almost 200,000 health-related websites, out of which about 42,000 are active internet pharmacies. Only 0.5% of them are legitimate, while 96.7% are not legitimate. The remaining 2.8% are pharmacies defined as potentially legitimate. This class is represented by pharmacies that do not fully adhere to the PharmaVerComp policies, but are probably not *illegitimate*. In this database, the examples are manually labeled by experts in the sector (i.e., human reviewers), and constitutes our ground truth.

We worked on two different instances of this database, generated with a six months difference, which were provided by PharmaVerComp. The summary statistics of these two instances are provided in Table 1. The intersection between the *illegitimate* instances of Dataset 1 and Dataset 2 is empty, i.e., in Dataset 2 we have 1275 different *illegitimate* domains. The two datasets contain the same *legitimate* instances, but crawled in different periods of time. We used crawler4j¹⁰ in order to crawl each one of the domains in the two datasets, without depth limit, but for a maximum of 200 pages. In our analysis we use Dataset 1 as *base* dataset to test our algorithms, while we use Dataset 2 to inspect how our models evolve over time, i.e., how models built on “old” data behave in dealing with “new” data.

We observe that the two classes in both datasets are strongly imbalanced. In order to cope with this situation, we can use *undersampling*, which modifies the frequencies of the two classes by randomly removing examples belonging to the majority class in the training set, until the minority class reaches the same percentage as the majority class. The other technique we used is *SMOTE* [9]. SMOTE is an oversampling technique in which the minority class is oversampled by creating “synthetic” examples. Examples are created operating in “feature space” rather than in “data space”, the opposite of what happens in oversampling with replacement. In our experiments we trained our classifiers using the natural class distribution as well as the ones generated using these two sampling techniques. Then, for each classifier, we took the one which offered the best results.

6.2 Evaluation Measures

In the following, we call “positive” the *legitimate* class, and “negative” the *illegitimate* class. With the notions TN , TP , FN , FP we denote respectively the number of true negatives, true positives, false negatives and false positive. Based on those, the evaluation measures we use are the following.

Overall Accuracy. Overall accuracy is the general correctness of the classifier and it is calculated as the sum of correct classified

instances divided by the total number of instances:

$$Acc = \frac{(TP + TN)}{(TP + TN + FP + FN)}.$$

Note that in the case of imbalanced classes this indicator does not provide a very good evaluation measure. In fact, given the actual distribution of our dataset (12% *legitimate*, 88% *illegitimate*), a simple strategy of guessing the majority class would have an accuracy of 88%, but, of course, such a kind of classifier does not help us to distinguish between *illegitimate* and *legitimate* pharmacies.

Precision. Precision is a measure of the accuracy provided for a specific class. It is defined as the number of correct classified instances for a specific class divided by the total number of instances for that class. For example, the precision for the *legitimate* class is computed as:

$$Precision_{legitimate} = \frac{TP}{TP + FP}.$$

Recall. Recall measures how many examples of one class are classified correctly. The recall of the *legitimate* class is computed as:

$$Recall_{legitimate} = \frac{TP}{TP + FN}.$$

Due to the imbalance of the two classes, we expect a good classifier to have a high *illegitimate* precision and an high *legitimate* recall. This would mean that the classifier is able to correctly identify the *legitimate* examples.

Area Under ROC Curve. The ROC curve is drawn by plotting the *False Positive Rate* ($FPR = \frac{FP}{TN+FP}$) against the *True Positive Rate* ($TPR = \frac{TP}{TP+FN}$) of the classifier, at various threshold settings. The ideal point on this curve would be on the top left corner, meaning that all the positive examples are classified correctly, and no negative examples are classified as positive. The area under ROC curve is a useful measure, especially in the case of imbalanced datasets.

Pairwise Orderedness. For what concern the second problem, we rely on a measure generally adopted in ranking problems, *pairwise orderedness*, which is an indicator of the number of “violations” of the ordered property in a list. First of all we define a function I as follows:

$$I(p, q) = \begin{cases} 1 & \text{if } rank(p) \geq rank(q) \text{ and } O(p) < O(q) \\ 1 & \text{if } rank(p) \leq rank(q) \text{ and } O(p) > O(q) \\ 0 & \text{otherwise} \end{cases}$$

This function give 1 if and only if a *illegitimate* pharmacy receives an equal or higher score than a *legitimate* pharmacy. Then, we evaluate our ranking computing the fraction of the pairs for which there is not such a violation:

$$pairord(\mathcal{X}) = \frac{|\mathcal{X}| - \sum_{(p,q) \in \mathcal{X}} I(p, q)}{|\mathcal{X}|},$$

where \mathcal{X} is the set of all the pairs of pharmacies (p, q) , $p \neq q$, in the set $\mathcal{P} \setminus \mathcal{P}_0$. If *pairord* is equal to 1 there are no violations in the pairs and we have all the *legitimate* pharmacies at the top of the list and all the *illegitimate* ones at the bottom.

6.3 Classification Results

We ran experiments on Dataset 1 in order to test the methods described in Sections 4.1 and 4.2. In particular, we trained different classifiers with several combinations of text and network features. Below we present the results of the text and network

¹⁰<https://github.com/yasserg/crawler4j>

classification, and ensemble classification, where we combine both.

For all results, we computed the corresponding confidence interval, which indicates the reliability of the results. We used a confidence level of $100(1 - \alpha)\%$ with $\alpha = 0.05$ (i.e., confidence level 95%). In all cases, the confidence intervals for our classifiers are very small (less than 1%): this means that the classifiers are stable, with the results of each fold being very close to the mean.

6.3.1 Text Classification. :

Text classifiers were trained with different text representation techniques on different subsets of the data. We employed 3-fold cross validation, where two folds were used for training and the third for testing. In the case of N-Gram Graphs, we randomly selected half of the training instances to build the class graph [13]. Hence, we compared these class graphs with all our instances (training and test). The N-Gram Graphs library we used to build the graphs and compute similarities is JInsect¹¹, while for the implementations of the classifiers we relied on Weka [16].

Table 2 lists the abbreviations concerning the classifiers and the sampling techniques employed in our experiments. We performed various tests with all combinations among classifiers and sampling techniques. However, due to space requirements, for each classifier we present only the sampling technique that performed best.

The overall accuracy with the TF-IDF representation are reported in Table 3. In all cases, accuracy is above 88%, with the best performers reaching 99%. However, as we can observe in Table 4, the J48 classifier has low *legitimate* recall for small subsamples of data. (As we already explained in Section 6.2, overall accuracy is not enough to properly evaluate a classifier in an imbalanced classes context.)

Increasing the number of words considered in the subsample generally results in better performance. SVM is the classifier that performs the best in terms of overall accuracy. Also for what concerns *legitimate* precision and *illegitimate* recall, the best performer is SVM (refer to Tables 4 and 5). It is interesting to note the inverse trend of NBM, whose efficacy decreases, especially in *legitimate* precision, as we consider larger term subsets. As expected, *illegitimate* precision is generally high, with all values above 93%. This derives directly by the imbalance of the two classes. In fact, since we have much less *legitimate* than *illegitimate* examples, if the classifier put some *legitimate* instances in the wrong class, this does not heavily affect the recall of the *illegitimate* class.

The AUC ROC curve, which is more robust to the case of imbalanced classes, is shown in Table 6. NBM is the winner in all cases considered. We note that SVM, which offers good results in terms of *legitimate* precision and *illegitimate* recall, does not have high AUC ROC values, especially for small subsets of terms. Another observation is that the choice of the sampling technique makes almost no difference for NBM and SVM. Instead, for J48, the sampling technique leads to substantial variations in performance. In particular, SMOTE is the sampling technique that offered the best results. Similar observations have also been documented in previous studies [8, 10].

We performed the same experiments using N-Gram Graphs, in order to compare the two text representation techniques. For N-Gram Graphs we do not use sampling, because of the nature of this representation.

Abbreviation	Description
NBM	Naïve Bayesian Multinomial
NB	Naïve Bayesian
SVM	Support Vector Machines
J48	Java implementation of C4.5 algorithm
MLP	Multilayer perceptron (Artificial Neural Networks)
NO	No sampling technique used
SUB	Subsampling
SMOTE	Oversampling with SMOTE algorithm

Table 2: Abbreviations

		#Terms				
		100	250	1000	2000	All
NBM	NO	0.97	0.97	0.96	0.96	0.95
SVM	NO	0.97	0.99	0.99	0.99	0.99
J48	SMOTE	0.89	0.92	0.93	0.96	0.95

Table 3: TF-IDF - Overall Accuracy

			#Terms				
			100	250	1000	2000	All
Recall	NBM	NO	0.90	0.98	0.98	0.97	0.96
	SVM	NO	0.76	0.90	0.93	0.92	0.95
	J48	SMOTE	0.57	0.61	0.71	0.83	0.78
Precision	NBM	NO	0.84	0.81	0.78	0.78	0.72
	SVM	NO	0.96	0.98	0.97	0.98	0.98
	J48	SMOTE	0.58	0.71	0.76	0.83	0.82

Table 4: TF-IDF - legitimate recall and precision

			#Terms				
			100	250	1000	2000	All
Recall	NBM	NO	0.98	0.97	0.96	0.96	0.94
	SVM	NO	0.99	0.99	0.99	0.99	0.99
	J48	SMOTE	0.94	0.96	0.97	0.97	0.98
Precision	NBM	NO	0.98	0.99	0.99	0.99	0.99
	SVM	NO	0.97	0.98	0.99	0.98	0.99
	J48	SMOTE	0.94	0.94	0.96	0.98	0.97

Table 5: TF-IDF - illegitimate recall and precision

		#Terms				
		100	250	1000	2000	All
NBM	NO	0.99	0.99	0.99	0.99	0.98
SVM	NO	0.88	0.95	0.97	0.96	0.97
J48	SMOTE	0.77	0.79	0.83	0.87	0.88

Table 6: TF-IDF - Area Under ROC Curve

		#Terms				
		#100	#250	#1000	#2000	All
NB	NO	0.89	0.89	0.94	0.95	0.92
SVM	NO	0.92	0.95	0.97	0.96	0.93
J48	NO	0.95	0.96	0.96	0.95	0.96
MLP	NO	0.97	0.98	0.98	0.99	0.99

Table 7: N-Gram Graphs - Classifiers Accuracy

			#Terms				
			#100	#250	#1000	#2000	All
Recall	NB	NO	0.53	0.48	0.63	0.70	0.60
	SVM	NO	0.43	0.61	0.77	0.73	0.60
	J48	NO	0.76	0.80	0.83	0.79	0.87
	MLP	NO	0.90	0.89	0.94	0.95	0.97
Precision	NB	NO	0.59	0.58	0.93	0.91	0.82
	SVM	NO	0.99	0.98	0.97	0.98	0.91
	J48	NO	0.87	0.91	0.88	0.83	0.89
	MLP	NO	0.88	0.93	0.94	0.95	0.94

Table 8: N-Gram Graphs - legitimate recall and precision

¹¹<http://sourceforge.net/projects/jinsect>

			#Terms				
			#100	#250	#1000	#2000	All
Recall	NB	NO	0.94	0.95	0.99	0.99	0.98
	SVM	NO	0.99	0.99	0.99	0.99	0.99
	J48	NO	0.98	0.99	0.98	0.98	0.98
	MLP	NO	0.98	0.99	0.99	0.99	0.99
Precision	NB	NO	0.93	0.92	0.95	0.96	0.93
	SVM	NO	0.92	0.94	0.97	0.96	0.93
	J48	NO	0.96	0.97	0.98	0.97	0.98
	MLP	NO	0.98	0.98	0.99	0.99	0.99

Table 9: N-Gram Graphs - *illegitimate* recall and precision

			#Terms				
			#100	#250	#1000	#2000	All
NB	NO		0.90	0.91	0.94	0.92	0.95
SVM	NO		0.71	0.81	0.88	0.86	0.80
J48	NO		0.93	0.92	0.91	0.88	0.95
MLP	NO		0.99	0.99	0.99	0.99	0.99

Table 10: N-Gram Graphs - Area Under ROC Curve

The overall accuracies are shown in Table 7. MLP (Artificial Neural Networks) is the classifier that offers the best accuracy results. MLP is also the winner when we consider *legitimate* recall and *illegitimate* precision, as well as the AUC ROC, though similarly to the TF-IDF case, SVM gives better results for *illegitimate* recall and *legitimate* precision (see Tables 8 and 9). Note that J48 is the second best classifier.

When we compare the results obtained by the two text representation techniques, we realize that they perform very close to one another. Nevertheless, TF-IDF has a small edge when compared to N-Gram Graphs, since it leads to slightly better *legitimate* recall and AUC ROC values for “small” documents, which are easier to handle.

The conclusion of these experiments is that the use of text classification in the process of internet pharmacy verification leads to good results, independently of the text representation technique used. The reason for such good performance resides on the fact that online *legitimate* and *illegitimate* pharmacies behave very differently when selling products.

Taking a close look at the most frequent terms used by *illegitimate* websites, we noticed that words like “viagra”, “cialis” and “no prescription” appear more frequently compared to *legitimate* pharmacies, which usually target a more broad and educated audience. Therefore, our classifiers built on top of the text representations of the pharmacy websites (using TF-IDF, and N-Gram Graphs) are in general able to recognize these differences and correctly predict the class of new instances.

6.3.2 Network Classification. :

In Table 11, we report the ten most linked-to websites by *legitimate* and *illegitimate* pharmacies. We observe that the most linked-to websites we find in the *legitimate* list are the two major social networks, Facebook and Twitter. This is in accordance with a previous study, which claims that *illegitimate* online pharmacies have fewer store presence features than *legitimate* pharmacies [23].

We can also find in the *legitimate* list many government websites that are not present in its *illegitimate* counterpart. For example, fda.gov, which is an agency responsible for protecting and promoting public health, is the third most linked-to website among the *legitimate* examples, while it is not even present in the *illegitimate* list. This is also the case for other government websites, like the National Institute of Health (nih.gov), and the Centers for Disease Control and Prevention (cdc.gov).

pointed by <i>legitimate</i> website	pointed by <i>illegitimate</i> website
facebook.com	wikipedia.org
twitter.com	wordpress.org
fda.gov	drugs.com
google.com	securebilling-page.com
youtube.com	rxwinners.com
nih.gov	google.com
adobe.com	providesupport.com
cdc.gov	euro-med-store.com
doubleclick.net	statcounter.com
nabp.net	cipl.com

Table 11: Websites pointed to by *legitimate* and *illegitimate* pharmacies (top 10)

Classifier	Overall Accuracy	AUC ROC
NB	0.96	0.95

Table 12: Network - Overall Accuracy and AUC ROC

	<i>legitimate</i> precision	<i>legitimate</i> recall	<i>illegitimate</i> precision	<i>illegitimate</i> recall
NB	0.904	0.732	0.966	0.990

Table 13: Network - precision and recall

On the other hand, we note that the two most linked-to websites in the *illegitimate* list are not directly related to the health sector (wikipedia.org and wordpress.org). Furthermore, in the *illegitimate* list there are some websites that are themselves classified as *illegitimate* pharmacies (e.g., rxwinners.com)¹² This fact is supported by many studies, which report networks of *illegitimate* pharmacies connected together in an affiliated way, where there is a central website and multiple other sites link to it¹³.

For the network experiments, we used the same settings as for the text classification. The dataset is divided into 3 folds (2 train, 1 test) and each experiment is repeated three times, changing the folds, according to cross-validation. Note that the two folds used for training represent the initial seed \mathcal{P}_0 . In the set of graph nodes V , we assigned 1 to those nodes that represent *legitimate* pharmacies in \mathcal{P}_0^+ , 0 to the others. We use the scores computed by TrustRank algorithm to train and test the classifiers, and the Naïve Bayes as the base classifier.

The overall accuracy and the AUC ROC are summarized in Table 12. The overall accuracy is around 96%, that is fairly close to the case of text classification, but for what concerns the AUC ROC curve the result is significantly worse. This is reflected also to *legitimate* recall, shown in Table 13, which is around 0.73, while for the other measures the method exhibits quite good results.

Given these results, we conclude that network analysis offers good performance in terms of *illegitimate* precision and recall, and could be used to assess the *legitimacy* of a pharmacy, even though it does not reach the level of confidence provided by the text analysis method.

6.3.3 Ensemble Classification. :

In order to enhance our results, we also combine the two analyses techniques, building a single model that embodies the characteristics of both the text and the network. In order to implement

¹²*illegitimate* status verified via the LegitScript public interface (<http://www.legitscript.com>).

¹³<http://legitscript.com/research>

	Acc.	<i>legitimate</i>		<i>illegitimate</i>		AUC ROC
		Rec.	Prec.	Rec.	Prec.	
Ensem. Sel.	0.96	0.92	0.96	0.99	0.99	0.99
Neural (Text)	0.98	0.94	0.94	0.99	0.99	0.99
NB (Network)	0.95	0.73	0.90	0.99	0.97	0.95

Table 14: Ensemble Classification Results

			<i>pairord</i>
TF-IDF	NBM	NO	0.998
	SVM	NO	0.999
	J48	SMOTE	0.994
N-Gram Graph			0.998

Table 15: Ranking using TF-IDF and N-Gram Graphs

this approach, we relied on “Ensemble Selection” [4], which is a method for constructing ensembles from libraries of models. We used an implementation of “Ensemble Selection” available in Weka. According to the process explained in [4], the library of models was trained, building a new model with standard parameters.

The results we obtained are presented in Table 14, comparing ensemble selection with the two best single models on text and network. For simplicity we report only the results considering subsamples of 1000 words; the other cases exhibit very similar results.

We note that ensemble selection increases the overall accuracy, *legitimate* precision and *illegitimate* recall when compared to the other classifiers. This results in a higher AUC ROC value, as well, which means that our ensemble classifier is the preferred method for this task.

6.4 Ranking Results

In Table 15, we summarize the ranking results for both the TF-IDF and the N-Gram Graph representations. As we expected, the results reflect the trend observed in the classification results. The best ranking is the one that is computed with the SVM classifier. Also the other classifier results follow the patterns highlighted in Section 6.3, with SVM and NBM performing better than J48.

As part of the ranking analysis, we performed an analysis of the *legitimate* and *illegitimate* outliers, i.e., the *illegitimate* examples that appear high in our ranking, and the *legitimate* examples that obtained poor score and appear at the bottom of the list. Performing a manual analysis on such examples gives us the possibility to check which *illegitimate* pharmacies are able to fool our system. Moreover, these insights could be very useful in helping *legitimate* pharmacies to better market themselves.

We extracted a subset of the *legitimate* and *illegitimate* outliers, and provided those to PharmaVerComp, in order to obtain feedback about their characteristics. The domain experts pointed out that *illegitimate* outliers, are in general not part of any *illegitimate* networks¹⁴. On the other hand the *legitimate* outliers are the pharmacies that offer new prescriptions, while the majority of them simply give the possibility to refill existing prescriptions.

6.5 Model Evolution over Time

As previously mentioned, we also want to test the behavior and robustness of our models over time. Given the good performance obtained with Dataset 1, we are now interested in evaluating how

¹⁴We recall that *illegitimate* examples are very likely to belong to *illegitimate* networks (see Section 6.3.2).

		Old-Old		New-New		Old-New	
		#Terms					
		250	1000	250	1000	250	1000
NBM	NO	0.99	0.99	0.99	0.99	0.99	0.99
SVM	NO	0.95	0.97	0.96	0.96	0.90	0.93
J48	SMOTE	0.79	0.83	0.80	0.84	0.74	0.78

Table 16: TF-IDF - Model over Time - Area Under ROC Curve

		Old-Old		New-New		Old-New	
		#Terms					
		250	1000	250	1000	250	1000
NBM	NO	0.81	0.78	0.73	0.74	0.61	0.57
SVM	NO	0.98	0.97	0.98	0.96	0.95	0.93
J48	SMOTE	0.71	0.76	0.67	0.79	0.66	0.51

Table 17: TF-IDF - Model over Time - *legitimate* Precision

much our models are affected by time. In particular, we want to answer the following two questions:

- (1) Will models computed on the new dataset (Dataset 2) get the same performance as the models computed on the old dataset (Dataset 1)?
- (2) Are models trained with the old data still valid on the new data?

The answer to the first question will give us the opportunity to evaluate the robustness of the model. The answer to the second question, will evaluate the validity of the proposed models over time, and test whether it is necessary to train the models often, or not.

In the following analysis, we only consider the classification part of the proposed approach. Note that ranking models are derived directly from the classifiers. We report here the performance for the two most meaningful classification measures for our problem, that is, AUC ROC and *legitimate* precision.

As we have seen, the first one offers a good overall indication of how well our classification process works, while the second measure is very sensible due to the small number of *legitimate* examples. Moreover, we focus on results for the Term Vector with TF-IDF weights and subsets of 250 and 1000 words.

6.5.1 New model with new data. :

In order to answer the first question posed above, we ran the same experiments conducted in Section 6.3, using Dataset 2. The results are used to verify if our models are effective even when applied to a new test dataset, despite the fact that *illegitimate* pharmacies appear in and disappear from the web at a relatively high rate.

We report the results in Tables 16 and 17. To do the comparison we report also the results obtained with the old dataset, namely, Dataset 1. In particular, we indicate with “Old-Old” the results obtained when computing and testing models on Dataset 1, and with “New-New” the results obtained by building and testing models on Dataset 2. We observe that the two models achieve almost the same performance for both measures. The conclusion of this analysis is that our approach is stable in analyzing different datasets that follow the natural distribution of instance classes.

6.5.2 Old model with new data. :

The answer to the second question will help us understand whether or not we need to adapt our models to pharmacy behavior changes. In particular, we expect that pharmacies change their text content and their relationship with other websites over

time. This may affect our approach, which is strongly based on these two factors.

We tested models computed on old dataset, i.e., Dataset 1, on the new Dataset 2. Recall that these two datasets were crawled with a difference of six months. In this period of time, online pharmacies may have changed their characteristics, especially the *illegitimate* pharmacies, since they may be closed by the inspection authorities.

The results of this analysis are presented in Tables 16 and 17 (column Old-New). We can observe that there is a small reduction in *legitimate* precision is evident, while the AUC ROC value remains almost the same.

The conclusion of this experiment is that our model is fairly robust over time. However, it has some problems related to the *legitimate* accuracy measure. In turn, this means that re-training and maintenance of the model is necessary to ensure good quality of results, though, it is not necessary that this re-training takes place very often.

7 CONCLUSIONS AND FUTURE WORK

In this paper, we proposed solutions for the problem of automatic internet pharmacy verification, which is becoming an increasingly relevant and important problem, gaining attention from both the public and the private sectors.

We formalized two different problems: first, a binary classification problem, where we define two classes, *legitimate* and *illegitimate*, and we classify the online pharmacies in one of them; and second, a ranking problem, where we seek a *totally ordered set* that defines a ranking among pharmacies. We then described solutions for both these problems, based on features that are relevant to the text and the network structure of online pharmacies. These are the first solutions that have been proposed in the literature in order to address the internet pharmacy verification problem.

We experimentally validated the effectiveness of our approach using two real datasets from two different time periods. The experiments demonstrate that the proposed algorithms can very accurately perform the classification and ranking tasks, and that the models we use are fairly robust over time. Our results confirm that our system could be effectively employed in the process of internet pharmacy verification, as well as in other similar tasks, offering considerable assistance to the human analysts dealing with such real-world problems.

As part of future work, we intend to extend our algorithms across two dimensions: (a) include in our network analysis non pharmacy websites that point to pharmacies, as well as consider websites at distances greater than one to our working set, and (b) study and evaluate classification schemes with combined (network and text), or additional features. In both cases, the aim will be to employ a richer input, and therefore to improve the performance of the algorithms.

Moreover, we plan to apply the proposed techniques to other domains of electronic commerce, where it will be possible to create publicly available datasets that can serve for making further progress in this area.

REFERENCES

- [1] Charu C. Aggarwal and ChengXiang Zhai. A survey of text classification algorithms. In Charu C. Aggarwal and ChengXiang Zhai, editors, *Mining Text Data*. Springer, 2012.
- [2] Rasim M. Alguliev, Ramiz M. Aliguliyev, and Saadat A. Nazirova. Classification of textual e-mail spam using data mining techniques. *Appl. Comp. Intell. Soft Comput.*, January 2011.
- [3] Roger Bate and Kimberly Hess. Assessing website pharmacy drug quality: Safer than you think? *PLoS ONE*, 5(8), 08 2010.
- [4] Rich Caruana, Alexandru Niculescu-Mizil, Geoff Crew, and Alex Ksikes. Ensemble selection from libraries of models. In *ICML '04*, 2004.
- [5] Carlos Castillo, Debora Donato, Aristides Gionis, Vanessa Murdock, and Fabrizio Silvestri. Know your neighbors: Web spam detection using the web topology. In *SIGIR '07*, 2007.
- [6] William B. Cavnar and John M. Trenkle. N-gram-based text categorization. In *SDAIR-94*, 1994.
- [7] Soumen Chakrabarti, Byron Dom, Rakesh Agrawal, and Prabhakar Raghavan. Scalable feature selection, classification and signature generation for organizing large text databases into hierarchical topic taxonomies. *The VLDB Journal*, 7(3), August 1998.
- [8] Nitesh V. Chawla. C4.5 and imbalanced data sets: investigating the effect of sampling method, probabilistic estimate, and decision tree structure. In *ICML '03*, 2003.
- [9] N.V. Chawla, K.W. Bowyer, L.O. Hall, and W.P. Kegelmeyer. Smote: Synthetic minority over-sampling technique. *JAIR*, 16, 2002.
- [10] Chris Drummond and Robert C Holte. *C4.5, Class Imbalance, and Cost Sensitivity: Why Under-sampling Beats Over-sampling*. 2003.
- [11] Susan Dumais and Hao Chen. Hierarchical classification of web content. In *ACM SIGIR, SIGIR '00*, 2000.
- [12] George Giannakopoulos, Vangelis Karkaletsis, George Vouros, and Panagiotis Stamatopoulos. Summarization system evaluation revisited: N-gram graphs. *TASLP*, 5(3), October 2008.
- [13] George Giannakopoulos, Petra Mavridi, Georgios Paliouras, George Papadakis, and Konstantinos Tserpes. Representation models for text classification : a comparative analysis over three web document types. In *WIMS 2012*. ACM, 2012.
- [14] George Giannakopoulos and Themis Palpanas. Revisiting the effect of history on learning performance: the problem of the demanding lord. *Knowl. Inf. Syst.*, 36(3):653–691, 2013.
- [15] Zoltan Gyongyi, Hector Garcia-Molina, and Jan Pedersen. Combating web spam with trustrank. Tr, Stanford InfoLab, March 2004.
- [16] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The weka data mining software: an update. *SIGKDD Explor. Newsl.*, 11(1), November 2009.
- [17] J.Pierre. On the automated classification of web sites, 2001.
- [18] Sepandar D. Kamvar, Mario T. Schlosser, and Hector Garcia-Molina. The eigentrust algorithm for reputation management in p2p networks. In *WWW'03*, 2003.
- [19] Ahmed Khorsi. An overview of content-based spam filtering techniques. *Informatica (Slovenia)*, 31(3), 2007.
- [20] Vijay Krishnan and Rashmi Raj. Web spam detection with anti-trustrank. In *AIRWeb*, 2006.
- [21] LegitScript. No prescription required: Bing.com prescription drug ads. Technical report, LegitScript, 2009.
- [22] Nektarios Leontiadis, Tyler Moore, and Nicolas Christin. Pick your poison: pricing and inventories at unlicensed online pharmacies. In *ACM EC'13*, 2013.

- [23] Tamilla Mavlanova and Raquel Benbunan-Fich. What does your online pharmacy signal? a comparative analysis of website trust features. In *HICSS*, 2010.
- [24] Katsiaryna Mirylenka, George Giannakopoulos, Le Minh Do, and Themis Palpanas. On classifier behavior in the presence of mislabeling noise. *Data Min. Knowl. Discov.*, 31(3):661–701, 2017.
- [25] A. Ntoulas, M. Najork, M. Manasse, and D. Fetterly. Detecting spam web pages through content analysis. In *WWW*, 2006.
- [26] World Health Organization. Counterfeit medicines. Technical report, World Health Organization, 02 2006.
- [27] G. Orizio, A. Merla, J.P. Schulz, and U. Gelatti. Quality of online pharmacies and websites selling prescription drugs: A systematic review. *JMIR*, 13(3), 2011.
- [28] G. Orizio, P. Schulz, S. Domenighini, L. Caimi, C. Rosati, S. Rubinelli, and U. Gelatti. Cyberdrugs: a cross-sectional study of online pharmacies characteristics. *EJPH*, 2009.
- [29] B. Raskutti and A. Kowalczyk. Extreme re-balancing for svms: A case study. *SIGKDD*, 6(1), June 2004.
- [30] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., 1986.
- [31] Fabrizio Sebastiani. Machine learning in automated text categorization. *ACM Comput. Surv.*, 34(1), March 2002.
- [32] Aixin Sun, Ee-Peng Lim, and Ying Liu. On strategies for imbalanced text classification using svm: A comparative study. *Decis. Support Syst.*, 48(1), December 2009.
- [33] Mikalai Tsytsarau and Themis Palpanas. Managing diverse sentiments at large scale. *IEEE Trans. Knowl. Data Eng.*, 28(11):3028–3040, 2016.
- [34] Mikalai Tsytsarau, Themis Palpanas, and Kerstin Denecke. Scalable detection of sentiment-based contradictions. In *International Workshop on Knowledge Diversity on the Web (DiversiWeb), in conjunction with the World Wide Web Conference (WWW)*, 2011.
- [35] G. Weiss and F. Provost. The effect of class distribution on classifier learning: An empirical study. Technical report, 2001.
- [36] Y. Zhang, D. Zhang, M.M. Hassan, A. Alamri, and L. Peng. Cadre: Cloud-assisted drug recommendation service for online pharmacies. *MONET*, 20(3), 2015.

RQL: Retrospective Computations over Snapshot Sets

Nikos Tsikoudis
 Brandeis University
 tsikudis@cs.brandeis.edu

Liuba Shrira
 Brandeis University
 liuba@cs.brandeis.edu

Sara Cohen
 Hebrew University, Jerusalem
 sara@cs.huji.ac.il

ABSTRACT

Applications need to analyze the past state of their data to provide auditing and other forms of fact checking. Retrospective snapshot systems that support computations over data store snapshots, allow applications using simple data stores like Berkeley DB or SQLite, to provide past state analysis in a convenient way. Current snapshot systems however, offer no satisfactory support for computations that analyze multiple snapshots. We have developed a Retrospective Query Language (RQL), a simple declarative extension to SQL that allows to specify and run multi-snapshot computations conveniently in a snapshot system, using a small number of simple mechanisms defined in terms of relational constructs familiar to programmers. We describe RQL mechanisms, explain how they translate into SQL computations in a snapshot system, and show how to express a number of common analysis patterns with illustrative examples. We also describe how we implemented RQL in a simple way utilizing SQLite UDF framework in a Berkeley DB data store using Retro page-level incremental snapshot system. Multi-snapshot computations running over page-level incremental snapshots bring up interesting performance issues that have not been studied before. We present the first study defining a performance envelope for multi-snapshot computations over page-level incremental snapshots.

1 INTRODUCTION

To provide auditing and other forms of claim checking more and more applications need to answer questions, often formulated after the fact, about the past states of their data. To free applications from the burden of managing past states on their own, data management systems need to run ad-hoc computations over past states of the objects they store.

Computations over past states have been long supported by temporal databases, used by applications in specialized domains but not used by general applications because of cost and performance penalty for in-production operation. More recently, cheap storage and interest in using past state analytics for in-production operation led to development of systems that integrate past state analytics in a database [7, 11, 13], and all major vendors today offer products providing OLTP and OLAP processing in a single system [16]. These products however are not a good match for Internet applications that store their data in simple key value stores such as Berkeley DB (BDB) [15] or SQLite [8] and need past state analysis for on-line historical claim checking or auditing. Today however, even applications using key value stores can support past state analysis using snapshot systems that support retro-spection, the ability of a data store to run queries over consistent snapshots of application past state as if they were the current state [22]. Retrospection makes it easy for programmers to provide expressive past state analysis since it allows to implement ad-hoc queries as general programs in the application language

using the application code base and then run these programs on the snapshots of interest. For example, Retro [21], a snapshot system of BDB, allows to analyze the state of BDB SQLite applications at a particular point in time simply by running SQLite queries over the corresponding BDB snapshot.

As convenient as it is to analyze a single point in time using a snapshot system, many analyses concern multiple data points. Retro and other snapshot systems come short when it comes to analyzing multiple snapshots. A programmer needs to write a C script that manually identifies snapshots of interest, queries each snapshot separately, manually collects the results, and then processes the results. This approach is cumbersome, error prone and onerous for a SQL programmer who needs to learn a new language. The programmer would much prefer to specify the computation in a declarative manner using the language of the application.

To help with programming the desired computation logic for multiple snapshot analysis, we have developed a Retrospective Query Language (RQL), a simple declarative extension to SQL that allows to specify and run multi-snapshot computations without the need to use a low-level script. RQL mechanisms combine in a modular way high-level relational constructs to express general SQL computations over arbitrary sets of past BDB SQLite application snapshots. The constructs specify in SQL the set of snapshots that identify the past states of interest, the computations over each snapshot, and the computations that process the results.

We describe RQL mechanisms and explain how each high-level mechanism translates into a SQL computation over multiple snapshots in the Retro snapshot system. We also show how to express a number of common analysis patterns with illustrative examples. We then describe how we implemented RQL in a simple way using SQLite UDF.

RQL programs bring to light important performance considerations that arise when programs compute over multiple snapshots. For one, RQL mechanisms allow to specify computations over arbitrary size sets of snapshots. The number of snapshots stored by a snapshot system such as Retro, only limited by available storage, can be very high given today's low storage costs. Each snapshot includes the entire state of the database. RQL program therefore can compute over potentially very large amounts of data. A programmer needs to know how much CPU, memory and I/O resources his program requires, especially in today's utility computing environments. Furthermore, an important performance consideration in the design of snapshot systems like Retro is to avoid interfering with the data store performance so that snapshots can be created at required frequency without blocking or disrupting in-production application performance. Retro snapshot system achieves this by using a low-cost copy-on-write technique that creates an incremental page-level snapshot representation with a compact snapshot index [22, 23]. Such representation is known to be slower to compute with but the slowdown is considered to be an acceptable trade-off to preserve in-production performance. The reason a computation runs slower over a snapshot and incurs higher resource costs compared to

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

the current database state is that the snapshot state needs to be assembled on-the-fly from the incremental representation. Inherently however, with a snapshot representation created using copy-on-write, consecutive snapshots share large parts of their state. An RQL program that iterates computations over multiple consecutive snapshots can therefore reduce its costs and improve performance by assembling such a shared state only once. The performance of programs computing over multiple *page-level copy-on-write* snapshots however has not been studied before.

To this end, we implemented RQL in SQLite BDB using Retro snapshot system and conducted an experimental study that characterizes the performance envelope of RQL programs. The goal of the study is to explain the performance in a way that is easy to understand by the programmer. Since the programmer specifies RQL mechanisms as a modular composition of relational constructs in SQLite, we relate the performance of RQL program to the performance of its SQL components, a cost that should be familiar to the programmer. Our experiments use workloads derived from the standard TPC-H benchmark to evaluate RQL performance so that even though our system is different from other past state systems, its performance is explained using a standard workload.

In summary, the paper makes the following contributions:

- RQL, a SQLite extension that allows to express computations over multiple snapshots in a convenient way in the language of the application. Our focus here is SQL extension but we believe a similar approach can be used for BDB applications in other languages since the BDB/Retro system is language-independent.
- An implementation of RQL system using SQLite UDF.
- A performance study of RQL programs including the first analysis explaining the performance of computations running over multiple page-level copy-on-write snapshots. While the performance results reported in our study are specific to our system, our performance analysis is more general and applies to other page-level copy-on-write snapshot systems.

The rest of the paper is organized as follows. Section 2 describes the RQL mechanisms, Section 3 outlines the salient points of RQL implementation using SQLite UDF, Section 4 briefly outlines the basic structure of the copy-on-write snapshot system Retro, providing the background for our performance analysis, Section 5 describes the experimental study, Section 6 considers the related work, Section 7 concludes.

2 RQL LANGUAGE

In this section we present RQL, the programming language for specifying SQL computations over sets of snapshots of past states in a data store. Our departure point is a transactional key value store with an integrated snapshot system that allows a SQL application to take snapshots of its state and subsequently run a SQL computation over a snapshot. Specifically, we assume the Retro snapshot system integrated with the BDB SQLite [21]. We first explain the snapshot computation model provided by Retro, and the concrete language constructs used by SQL programmers to create snapshots and to specify a program that runs over a snapshot.

Retro extends BDB/SQLite with a language construct that allows to declare a snapshot as part of normal transaction commit using the *BEGIN*; and *COMMIT WITH SNAPSHOT*; commands. The declaration command creates a transactionally consistent

l_userid	l_time	l_country
UserA	2008-11-09 13:23:44	USA
UserB	2008-11-09 15:45:21	UK
UserC	2008-11-09 15:45:21	USA

(a) Snapshot S1

l_userid	l_time	l_country
UserB	2008-11-09 15:45:21	UK
UserC	2008-11-09 21:33:12	USA

(b) Snapshot S2

l_userid	l_time	l_country
UserB	2008-11-09 15:45:21	UK
UserC	2008-11-09 21:33:12	USA
UserD	2008-11-11 10:08:04	UK

(c) Snapshot S3

Figure 1: LoggedIn table in snapshots 1-3

snap_id	snap_ts
1	2008-11-09 23:59:59
2	2008-11-10 23:59:59
3	2008-11-11 23:59:59

Figure 2: SnapIds table

persistent snapshot that includes the state of the entire database (e.g., tables, indexes, system catalogs). The snapshot reflects the modifications committed by the declaring transaction T, and all the transactions committed before T. The declaration permanently associates with the snapshot a unique snapshot identifier that names the snapshot. The identifier and a current timestamp are entered in a table with name *SnapIds*. Although Retro uses integer sequence numbers as snapshot identifiers internally, a programmer can associate meaningful snapshot names with the identifiers.

A query can run on a snapshot at any point following the snapshot declaration. To run a SQL program, such as "SELECT ..." over previously declared snapshot with identifier *sid*, the programmer simply specifies "SELECT AS OF *sid* ...". SQL queries and update transactions that do not declare snapshots remain unchanged by Retro.

Consider an example SQL application using a *LoggedIn* table that stores the users who are logged in a system, along with the time and the country from which each user has logged in. Whenever a user logs out, he is deleted from the table. Figure 3 shows a SQL program containing three consecutive snapshot declaration commands, (lines 1-2, lines 3-5, and lines 6-8), a snapshot query command that runs on the snapshot *sid* 1 (line 9), and the same query that runs on the current database state (line 10). Figure 1 shows the table state in three declared snapshots, and Figure 2 shows the *SnapIds* table. Note, that the state of *LoggedIn* table in the snapshot 2 declared in lines 3-5 does not include *UserA* since a snapshot reflects updates of the declaring transaction.

Retro makes it easy to analyze a single snapshot but has no support for analysis concerning multiple snapshots. In order to provide this functionality we propose RQL, a simple language for specifying computations over a set of Retro snapshots.

```

Declare snapshot S1
1. BEGIN;
2. COMMIT WITH SNAPSHOT;

Update table and declare snapshot S2
3. BEGIN;
4. DELETE FROM LoggedIn WHERE l_userid = 'UserA';
5. COMMIT WITH SNAPSHOT;

Update table and declare snapshot S3
6. BEGIN;
7. INSERT INTO LoggedIn (l_userid, l_time, l_country)
   VALUES ('UserD', '2008-11-11 10:08:04', 'UK');
8. COMMIT WITH SNAPSHOT;

Retrospective query
9. SELECT AS OF 1 * FROM LoggedIn;

Query on current state
10. SELECT * FROM LoggedIn;

```

Figure 3: Retro SQL example

RQL queries specify computations using a small set of basic computational mechanisms that compose relational constructs familiar to a SQL programmer. A computation defined by an RQL query iterates over a set of snapshots, runs a SQL query on each snapshot, collects results of the query and performs on the results different combining computations determined by the specific mechanism used.

In specifying the set of snapshots to iterate over, and the snapshot query to be executed in each iteration, RQL uses two auxiliary constructs, a *snapshot table* that holds all the declared snapshot identifiers and timestamps, referred to as *SnapIds*, and a function *current_snapshot* that provides the identifier of the snapshot used in the iteration where the current computation is performed.

The RQL mechanisms *Collate Data*, *Aggregate Data In Variable*, *Aggregate Data In Table* and *Collate Date Into Intervals* are best described operationally by explaining the SQL computation each one performs.

2.1 Collate Data

The RQL mechanism *Collate Data* collects records from multiple snapshots into a table.

```
CollateData(Qs, Qq, T)
```

Collate Data requires three parameters, the queries *Qs*, *Qq* and table name *T*. Query *Qs* returns a single column containing snapshot identifiers selected from the snapshot table *SnapIds*. The *Qs* output represent the snapshot set (interval) the programmer is interested in. Query *Qq* is applied to every snapshot in the snapshot interval. *T* is the name of the table into which we collate the results of every *Qq*.

For the first snapshot identifier *Sx* returned by the *Qs*, the mechanism issues "CREATE TABLE *T* AS *Qq*" within the snapshot *Sx*. For all the subsequent *Sy* returned by *Qs*, "INSERT INTO *T* *Qq*" is issued, within snapshot *Sy*.

The following example collects all the user_ids and the snapshot identifier of the snapshot they appear in.

```
CollateData("SELECT snap_id FROM SnapIds",
```

```
"SELECT DISTINCT l_userid, current_snapshot()
FROM LoggedIn", "Result")
```

Collate Data along with the snapshot table *SnapIds* and the *current_snapshot()* function provide a general language for implementing any kind of computation by issuing SQL queries on the Collate Data output. However, Collate Data can have a large footprint in terms of the memory required to hold its result, especially when the *Qs* set and *Qq* result set are large.

The two aggregation mechanisms we present next, allow to reduce the memory footprint for RQL computations that need to aggregate results over snapshots.

2.2 Aggregate Data In Variable

The first aggregation mechanism *Aggregate Data In Variable* applies an aggregate function on a single element across multiple snapshots.

```
AggregateDataInVariable(Qs, Qq, T, AggFunc)
```

In addition to the queries *Qs*, *Qq* and table name *T*, it requires an aggregate function as a parameter and it expects from *Qq* to return a single row and a single column. For the first snapshot identifier *Sx* returned by *Qs*, we execute *Qq* on snapshot *Sx* and save the single value in a variable *V1*. For all subsequent identifiers *Sy* returned by *Qs*, we issue *Qq* on snapshot *Sy*, save the single value in a variable *V2* and update *V1* as *AggFunc(V1, V2)*. Finally, we store the result in the table *T*.

The following example shows how we can count the number of snapshots in which a tuple appears. For example, we want to count the number of snapshots in which user *UserB* is logged in.

```
AggregateDataInVariable("SELECT snap_id FROM SnapIds",
"SELECT DISTINCT 1 FROM LoggedIn
WHERE l_userid = 'UserB', "Result", "sum")
```

In the next example, we want to find the first occurrence of the same user.

```
AggregateDataInVariable("SELECT snap_id FROM SnapIds",
"SELECT DISTINCT current_snapshot() FROM LoggedIn
WHERE l_userid = 'UserB' ", "Result", "min")
```

2.3 Aggregate Data In Table

The mechanism *Aggregate Data In Table* provides the ability to apply aggregate functions on records of multiple columns across snapshots.

```
AggregateDataInTable(Qs, Qq, T, ListOfColFuncPairs)
```

The additional required parameter is a list of pairs of column names and aggregate functions.

For the first snapshot identifier *Sx* returned by *Qs*, we create a table *T* and insert the *Qq* output. For all the subsequent identifiers *Sy* returned by *Qs* we issue the query *Qq* and for each record in its output we search in table *T* to find a tuple with the same values in columns not included in the *ListOfColFuncPairs*. If such a record exists we perform the required computation on the values in columns of *ListOfColFuncPairs*, otherwise we insert into *T* the record returned by *Qq*. Note, the *Aggregate Data In Table* queries can be considered as across time *GROUP BY* queries where the grouping columns are the columns of the *Qq* output not appearing in the *ListOfColFuncPairs*. So, for the aggregation across snapshots to be well defined, *Qq* should never return two records that coincide on all the values in the grouping columns.

The following example shows how we can find the first time that each user has logged in.

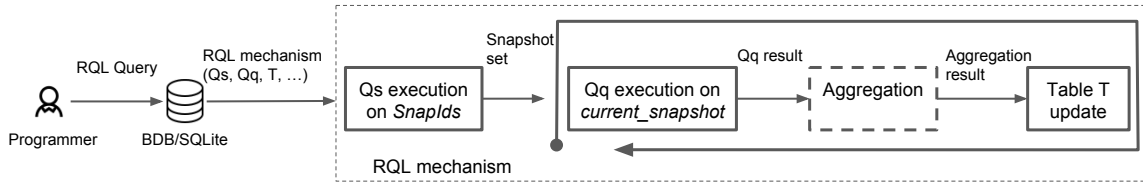


Figure 4: General structure of RQL computations

```
AggregateDataInTable("SELECT snap_id FROM SnapIds",
"SELECT DISTINCT l_userid, l_time FROM LoggedIn",
"Result", "(l_time,min)")
```

The next example shows how we can compute, for each country, what is the maximum number of users who are simultaneously logged in, from that country.

```
AggregateDataInTable("SELECT snap_id FROM SnapIds",
"SELECT l_country, COUNT(*) AS c FROM LoggedIn
GROUP BY l_country", "Result", "(c,max)")
```

In order for *Aggregate Data in Variable* and *Aggregate Data in Table* to work as described, we require the aggregate function to satisfy certain mathematical properties. Formally, the aggregate function must be definable by an abelian monoid (X, op, e) where X is the domain of values, op is an associative and commutative binary operation and e is the identity element. Most SQL aggregate functions e.g. min, max, count and sum, satisfy the requirement but some, e.g. average, and aggregations over distinct elements e.g. count distinct or sum distinct do not. Because average is widely used in SQL, our aggregation mechanisms implement a simple extension that support average as a special case. Aggregations over distinct elements can use the *Collate Data* mechanism to return a column containing the elements and then use SQL to perform the needed aggregation. Of course, such approach may not reduce the memory footprint of the result computation.

2.4 Collate Data Into Intervals

The mechanism *Collate Data Into Intervals* creates an alternative, potentially more compact snapshot data representation that resembles the traditional record lifetime representation used by temporal databases, and could be used by applications that expect this kind of representation. When the snapshots are taken frequently, it is likely that the same record appears in many consecutive snapshots. The mechanism collects records from multiple snapshots into intervals which indicate the lifetime of the records. This is achieved by creating two attributes in our result table and storing the *start_snapshot* and the *end_snapshot* for each record's lifetime.

```
CollateDataIntoIntervals(Qs, Qq, T)
```

Collate Data Into Intervals requires the same parameters as *Collate Data*. For the first snapshot identifier S_x returned by the Q_s , we create the table T and insert the records returned by Q_q with *start_snapshot* and *end_snapshot* for each record set to S_x . For all the subsequent S_y returned by Q_s we issue the query Q_q and for each record in its output we search in table T to find a tuple with the same values in all columns except the *start_snapshot* and *end_snapshot*. If a tuple exists and if its *end_snapshot* value is the same as the snapshot identifier of the previous iteration we update the *end_snapshot* to S_y , otherwise we insert a new tuple with *start_snapshot* and *end_snapshot* set to S_y .

The following example calculates the interval during which each user was logged in.

```
CollateDataIntoIntervals("SELECT snap_id FROM SnapIds",
"SELECT l_userid FROM LoggedIn", "Result")
```

Figure 4 provides the general structure of our RQL computations. The *Aggregation* part is bypassed in case of *Collate Data*.

3 IMPLEMENTATION

An RQL computation iterates (loops) over the snapshots in the snapshot set defined by the parameter query Q_s , and for each snapshot it executes a "loop body" that invokes the query Q_q on this snapshot, processing its results in a way specific to each mechanism. This section explains how we implement this computation in SQLite/BDB Retro system using SQLite UDF. We highlight the salient points of the implementation, including the cross snapshot iteration with constructs *current_snapshot()* and *SnapIds*, and the processing of snapshot query results.

We create the SQL program for RQL computation by composing the Q_s and Q_q query programs using the callback infrastructure provided by SQLite UDF. The infrastructure allows to interpose a UDF callback function on a SELECT statement so that the callback is invoked for each element of a set returned by the SELECT.

We define the "loop body" of our computation in a UDF callback function, providing for each mechanism a mechanism-specific callback, and iterate over snapshots by interposing the "loop body" callback on the SELECT statement for Q_s .

The following SQLite statement shows the general syntax used by our implementation for an RQL mechanism.

```
SELECT rql_udf (snap_id, Qq, T, ...)
FROM SnapIds WHERE ...;
```

By issuing this statement to SQLite, we achieve the iteration over the snapshot identifiers in the table *SnapIds* returned by the SELECT (i.e. Q_s), where for each returned snapshot identifier, SQLite invokes the "loop body" defined by the UDF callback *rql_udf*. Figure 5 shows the general structure of the resulting computation.

The UDF argument *snap_id* is filled at runtime by SQLite with values returned by Q_s in each iteration, the other parameters, including the string defining the Q_q query, the table name T , and additional parameters needed for the aggregation mechanisms are specified by the programmer.

Inside the "loop body" UDF, we treat the parameter *snap_id* as "loop index". The "loop body" UDF uses Retro to run the query Q_q on snapshot *snap_id* in every iteration it gets invoked.

To run on a snapshot *snap_id*, Retro requires a query to be in the form of "SELECT AS OF *snap_id* ...". Furthermore, the SQL program Q_q may include the function *current_snapshot()*, explained in Section 2, that denotes the snapshot identifier of the current iteration. Therefore, as a first step, our "loop body" UDF rewrites the Q_q , binding it to the value of "loop index" *snap_id*.

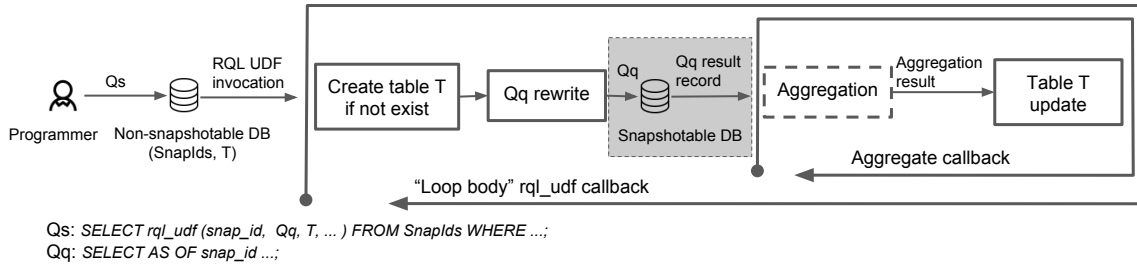


Figure 5: Implementation structure of RQL computations

The rewriting involves adding the "AS OF snap_id" extension, and replacing every occurrence of `current_snapshot()` function with the value of `snap_id`.

For example, for the iteration with `snap_id` value of `Si`, the following Qq query typed by the programmer:

```
SELECT DISTINCT current_snapshot() FROM LoggedIn
WHERE l_userid = 'UserB';
```

will be rewritten by the RQL UDF to take the following form:

```
SELECT AS OF Si DISTINCT Si FROM LoggedIn
WHERE l_userid = 'UserB';
```

After Qq is rewritten, the UDF issues it to SQLite, invoking the `sqlite3_exec` function from the SQLite API. The "loop body" UDF then proceeds to process the results across snapshots in a mechanism-specific manner, as shown in Figure 5. Note the shaded area depicts the Qq computation run by Retro within a single snapshot.

Consider an example *Collate Data* query specified below:

```
SELECT CollateData(snap_id,
"SELECT DISTINCT l_userid, current_snapshot() AS sid
FROM LoggedIn", "Result") FROM SnapIds;
```

Since the mechanism does not perform any computation besides Qq, its "loop body" UDF only utilizes the SQLite API. That is, *Collate Data* UDF callback executes SQL statements using the `sqlite3_exec` function to insert Qq results into the result table T.

The UDF callbacks that implement our aggregate mechanisms are more involved since they need to implement aggregation. Consider an example of *Aggregate Data in Variable* specified as follows:

```
SELECT AggregateDataInVariable(snap_id,
"SELECT DISTINCT current_snapshot() AS sid
FROM LoggedIn WHERE l_userid = 'UserB' ",
"Result", "min") FROM SnapIds;
```

In the first iteration, its "loop body" UDF creates a table `Result` with attribute `sid`. It then rewrites Qq and invokes the `sqlite3_exec` function from the SQLite API to run the Qq. One of the arguments of `sqlite3_exec` is a callback function which gets invoked for every tuple returned by the Qq, providing access to the returned tuple. In this callback function we implement the aggregate computation for *Aggregate Data In Variable*, directly following the specification in the previous section.

Consider next an example of *Aggregate Data in Table* specified as follows.

```
SELECT AggregateDataInTable(snap_id,
"SELECT l_country, COUNT(*) AS c FROM LoggedIn
GROUP BY l_country", "Result", "(c,max)") FROM SnapIds;
```

Here the first "loop body" iteration creates the table `Result` with attributes `l_country` and `c`. It then rewrites and issues Qq and for

every tuple returned by the Qq a callback function is invoked that inserts the tuple in the table `Result`. At the end of the first "loop body" iteration we also create an index on `Result` using as key the values in non-aggregating columns, in this case `l_country`. In subsequent iterations, for every record returned by the Qq, in the callback function we utilize the index to search in table `Result`. If a tuple with the same values in non-aggregating columns is present, we apply the aggregate functions on the columns specified in the `ListOfColFuncPairs` and update the record in the result table accordingly, otherwise we insert a new tuple.

We have also experimented with alternative *Aggregate Data in Table* implementation using a sort-merge based algorithm that turned out to be costlier.

For brevity, we omit the implementation details of *Collate Data Into Intervals*. It is implemented similarly to *Aggregate Data in Table* but instead of applying an aggregate function we check whether we need to update the record's lifetime or insert a new tuple in the result table.

Note that RQL mechanisms by default create the result table T as temporary non-snapshotable table. However, it can be created as persistent if the programmer decides otherwise.

We now briefly consider the construct *Snapids*. It is currently implemented at application level to support user friendly snapshot names. Also, it is stored in a separate SQLite database than application data because it is a non-snapshotable persistent table (whereas the rest of the data are snapshotable). Every time the application declares a snapshot and gets back the snapshot identifier, it inserts the identifier in the `SnapIds` along with a timestamp and any application meaningful information the programmer needs to later refer to the snapshot. The update operations on `SnapIds` table are transactional. Note, that concurrent updates to `SnapIds` table and RQL queries do not block each other since Retro runs snapshot queries as read-only snapshot transactions taking advantage of MVCC concurrency control in BDB, as we explain in section 4. Nevertheless, updating `Snapids` in a transaction adds overhead so we are currently working on an internal implementation to reduce this overhead.

4 RETRO SNAPSHOT SYSTEM

We briefly describe Retro, the snapshot system used in RQL. Our goal is to explain how the cost of snapshot query is impacted by the method of incremental page-level snapshot creation and indexing, and the different update workloads. The complete description of Retro system can be found in [21–23].

Retro snapshot system is implemented as a small set of modular extensions to the Berkeley DB transactional storage manager. The extensions interpose on transaction *commit*, *page flush*, *page fetch and recovery* operations. The implementation at the storage manager level allows to create transactionally consistent,

recoverable snapshots efficiently, without blocking application transactions, by exploiting the BDB MVCC concurrency control and recovery mechanisms [22].

The snapshot system interface supports two operations, *snapshot declaration* and *snapshot query*, that can be exposed to applications in a language-specific way. Section 2 presented the SQL interface. A Retro snapshot is a set of immutable logical data pages that reflect the entire consistent database state, including the catalog and indexes, at snapshot declaration point. This allows to run on a snapshot any database query q that could run in the database at the snapshot declaration point. The snapshots are captured using a page-level copy-on-write technique (COW) that copies out and saves snapshot pages incrementally as transactions commit modifications to pages, following a snapshot declaration. At transaction commit time, Retro identifies any page P that is modified for the first time following the declaration of a snapshot S and copies out the pre-modification state (pre-state) of P . The pre-state corresponds to the state of P as-of snapshot S . If this modification of P is also the first since an earlier declared snapshot S' , the pre-state is shared by S' .

Retro accumulates the copied-out pre-states in memory and writes them to an on-disk log-structured snapshot archive called *Pagelog* when the database flushes updates. The pre-states are indexed at low cost by simply recording a mapping that associates a snapshot page P with its *Pagelog* location. Retro writes the mappings to an on-disk log-structured list called *Maplog* [23]. When a snapshot S is needed, an efficient scan of *Maplog* allows to construct a snapshot page table $SPT(S)$ that maps every page P in snapshot S to its location in *Pagelog*. The scan length is $n \log(n)$ where n is the number of pages in the snapshot, independent of snapshot history length [23].

To run a query q on a snapshot S Retro interposes on the database page *fetch* operation. When q requests a page P , Retro looks up page location in $SPT(S)$ and fetches P from *Pagelog*, the same way q would fetch P from the database if it was running on the current database state.

Retro caches snapshot pages in a buffer cache along with the database pages and needs extra cache memory to hold the snapshot pages when running snapshot queries. While we expect the database pages to reside in memory given today's large memories, we do not expect snapshot pages to fully reside in memory because with long snapshot histories *Pagelog* can grow very large, limited only by the available disk space. For this reason, even with a large snapshot cache, when a query runs on a snapshot that has not been accessed recently, we expect the snapshot page cache hit rate to be low. The *I/O* cost of a snapshot query therefore depends on the number of pages it fetches from *Pagelog* when the page is not present in the cache.

Retro allows to run snapshot queries concurrently with current state queries and update transactions. It relies on the BDB concurrency control scheme MVCC to avoid snapshot queries from interfering with updates. Consider a snapshot query q that runs over snapshot S following the snapshot declaration. At this point the snapshot shares all its pages with the database. Consider a page P requested by q and a concurrent transaction T that modifies page P . Retro runs q as a read-only MVCC transaction relying on MVCC to provide q with the unmodified pre-state of P to the end of q , without interfering with T .

We now consider the amount of *Pagelog I/O* needed by a snapshot query. Consider a set of update transactions T_1, \dots, T_n that updates every page in the database following the declaration of snapshot S . We call such transaction set an overwrite cycle

of S . After the overwrite cycle of S completes, the entire state of S and all the snapshots declared before S , is copied out into *Pagelog*. A snapshot declared a long time ago is likely to have a complete overwrite cycle so a query running over old snapshots fetches all its pages not present in a cache from *Pagelog*. On the other hand, consider a snapshot S that has been declared recently. Some of the pages of the database likely have not been modified since snapshot S declaration, so its overwrite cycle is incomplete. If a page P has not been modified since snapshot S declaration, S shares P with the database. A snapshot query running on S and requesting a shared page P will fetch P from the database. Therefore, we expect a query running on recent snapshots to have less *Pagelog I/O*.

Finally, consider two snapshots S_1 and S_2 declared consecutively, and the set of update transactions T_1, \dots, T_k committed between S_1 and S_2 . Note that snapshot S_1 and S_2 share all their pages except those pages modified by T_1, \dots, T_k . Let $shared(S_1, S_2)$ be the set of pages shared by S_1 and S_2 , and $diff(S_1, S_2)$ be the set of pages that are not shared by S_1 and S_2 . Consider a snapshot query q running consecutively over two old snapshots S_1 and S_2 using a cache large enough to hold all the snapshot pages requested by q . If snapshots S_1 and S_2 have not been accessed for a long time, all pages of S_1 will need to be fetched from *Pagelog* but any page P in $shared(S_1, S_2)$ needs to be fetched from *Pagelog* only once since P will be in the cache after q runs on S_1 . On the other hand, any page Q in $diff(S_1, S_2)$ requested by q running on S_2 likely will need to be fetched from *Pagelog*. The size of $diff(S_1, S_2)$ will therefore determine the cache miss rate for q running on S_2 . The size of $diff(S_1, S_2)$ is determined by the transaction update workload, i.e. by how many pages the transactions modify, and by the frequency of snapshot declarations, i.e. how many transaction apart are the declarations S_1 and S_2 . If transactions modify many pages and snapshot declaration are infrequent, $diff(S_1, S_2)$ will be large, but if transactions modify few pages and snapshot declarations are frequent, $diff(S_1, S_2)$ will be small and S_1 and S_2 will share most of the pages.

5 PERFORMANCE EVALUATION

This section presents an experimental study that characterizes the performance of RQL computations. We aim to explain RQL performance in terms that are familiar to a SQL programmer. Since the programmer specifies an RQL query r by providing SQL programs Q_s, Q_q and the aggregation functions, we consider how the performance characteristics of these SQL programs impact the performance of r .

To explain the performance of RQL we need to characterize the costs of a computation that iterates over snapshots. The performance of a snapshot computation that runs over a stand-alone single snapshot has already been studied [21]. However, as we explain in Section 4 because of snapshot page sharing, a snapshot computation that runs as one of the RQL iterations can have a different performance than a snapshot computation that runs on a stand-alone snapshot. Our study evaluates the benefit of page sharing for different transaction update workloads and explains how the benefit of sharing depends on the properties of the snapshot set Q_s , and whether the snapshot computation Q_q is *I/O* or computation intensive. Our experiments also analyze the memory requirements of different RQL mechanism and quantify the memory benefits of the aggregation mechanisms.

Our experiments run our implementation of RQL in the Retro snapshot system integrated with BDB SQLite version 5.3.21.

Parameters	Notations	Description
Update Workload	UW15 UW30	Delete and insert 15K orders and their lineitem records per snapshot Delete and insert 30K orders and their lineitem records per snapshot
Query Qs	Qs_N	Query that determines the snapshot interval length N
Query Qq	Qq_io Qq_cpu Qq_collate Qq_agg Qq_int	SELECT COUNT(*) FROM orders WHERE o_orderstatus = 'O' ; SELECT SUM(l_extendedprice) AS revenue FROM lineitem, part WHERE p_partkey = l_partkey and p_type = 'STANDARD POLISHED TIN'; SELECT o_orderkey FROM orders WHERE o_orderdate < '[DATE]'; SELECT o_custkey, COUNT(*) AS cn, AVG(o_totalprice) AS av FROM orders GROUP BY o_custkey; SELECT o_orderkey, o_custkey FROM orders;
RQL UDF		CollateData (Qs, Qq, T) AggregateDataInVariable (Qs, Qq, T, AggFunc) AggregateDataInTable (Qs, Qq, T, ListOfColFuncPairs) CollateDataIntoIntervals (Qs, Qq, T)
Aggregate function		MIN, MAX, SUM, COUNT, AVG

Table 1: Parameters and notations

The hardware platform consists of 2 hexa-core Intel Xeon CPU clocked at 2.50 GHz with Hyper-Threading enabled, 2 Intel 400 GB SATA SSD and 64 GB of RAM. The operating system is Red Hat Enterprise Linux Server release 6.8 (Santiago), x86_64 and the file system is formatted with ext4.

RQL performance depends on the database update workload, the in-snapshot query workload, and the type of RQL computation that combines the results of the snapshot queries. We expect the in-snapshot queries to include both *native* queries the application runs in the current state, e.g. when performing auditing, and *ad-hoc* queries formulated after the fact, e.g. when performing fact checking. The native queries are more likely to have native indexes captured in the snapshot. The ad-hoc queries may need to create the indexes at RQL execution time. Our experiments use both native and ad-hoc queries.

The database we use for our experiments is a TPC-H database. TPC-H [6] is a standard decision support benchmark consisting of tables designed to be business relevant and the database schema includes tables with information about Customers, Orders, Lineitems, etc. We create the database by using the TPC-H *dbgen* tool to produce the initial state of the database with size of 1.4 GB (the default size) without additional indices. The size increases accordingly when indices are included.

In order to create a snapshot history, we utilize the TPC-H refresh functions which produce a set of order identifiers for deletion and a set of order records along with Lineitem records associated with the orders for insertion. Our update workload program receives as input the TPC-H refresh function output, updates the database by deleting and inserting a certain number of Orders and their Lineitem records and creates snapshots. Between two consecutive snapshot declarations a constant number of orders and their associated records are inserted and deleted making it easier to interpret the performance results and memory requirements.

We consider two update workloads that delete and insert different amount of data, as defined in Table 1. These update workloads generate different $diff(S1, S2)$, the amounts of non-shared data between two consecutive snapshots $S1$ and $S2$, and affect how frequently the database gets overwritten, i.e. the length of the

snapshot overwrite cycle. The UW30 overwrites the database every 50 snapshots while the UW15 overwrites every 100 snapshots, so the $diff(S1, S2)$ in UW30 is double the size of UW15.

We assume the current state database is memory resident, and the snapshot pages are stored in Pagelog on the SSD. We achieve the expected snapshot cache behavior by assuming the snapshot page cache is empty at the start of an RQL query, and assume the cache can hold the snapshot pages requested by a single RQL query, except when discussing memory costs.

For our experiments, we define custom queries that stress different RQL costs in both native and ad-hoc queries. Table 1 reports all the queries used throughout our performance evaluation. We explain the characteristics of each query when describing the experiments that use them. The reason we don't provide any experimental results using TPC-H queries is because their complexity makes them CPU intensive and does not allow us to stress and focus on a single RQL cost each time.

5.1 I/O intensive queries: Impact of snapshot sharing

Our first set of experiments considers the impact of snapshot sharing on the *I/O* costs of an RQL query for old and recent snapshots.

We first consider old snapshots. When all the snapshots included in the set defined by Qs are old, the first iteration fetches from the Pagelog disk all the pages it needs and is likely to fetch the highest number of pages compared to the subsequent iterations. We refer to the first iteration as *cold*, and to the subsequent iterations as *hot*. Note, the number of pages fetched by a cold iteration is identical to a stand-alone snapshot query and is determined by the code of Qq . The maximum number of pages potentially fetched by the subsequent *hot* iterations depends on two factors, the $diff(S1, S2)$ in the update workload explained in Section 4 and how far apart are the snapshots in the hot iterations, determined by the number of snapshots skipped in the Qs query. In the extreme case, if Qs defines a skip that exceeds the snapshot overwrite cycle length, the performance of a hot iteration will be no different than a performance of a cold iteration. We refer to an RQL query run where all iterations are cold

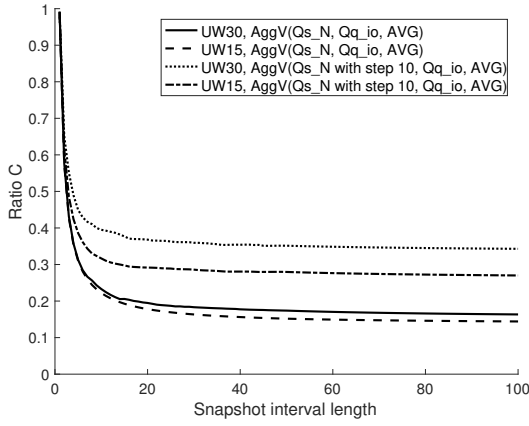


Figure 6: Ratio C with old snapshots: impact of sharing between snapshots.

as *all – cold*. In general, we expect sharing to improve the RQL query performance compared to *all – cold* but the amount of improvement depends on the snapshot interval length since for short intervals the performance of cold iteration may dominate.

The combined impact of sharing can be succinctly captured using a ratio C of latency of an RQL query r with a given Qq and Qs , to the latency of an *all – cold* run with the same number of snapshots.

Figure 6 shows the ratio C as the number of snapshots in the interval increases, for update workloads with different amount of sharing, UW30 and UW15 and different distance between consecutive iterations. To isolate the impact of sharing on total *I/O* costs we use a computationally light RQL Aggregate Data in Variable query with an *I/O* intensive and computationally light Qq_{io} . At each RQL iteration, Qq scans the table Orders and returns the number of open orders for the current snapshot and the RQL query computes the average number of open orders per snapshot.

Since all snapshots are old, the cost of the *all – cold* run remains constant. Sharing increases as we move from update workload UW30 to UW15 and when the number of skipped snapshots drops from 10 to 1. The ratio C drops with increased sharing reflecting the RQL query latency decrease compared to *all – cold* run. Overall, C is high for short intervals since sharing makes little difference as the cost of the first cold iteration dominates the RQL query latency. For sufficiently long intervals however (more than 20 snapshots) C converges to a constant as the cost of the cold iteration stops being the dominant cost, and the RQL query latency is fully determined by sharing. The first two bars in Figure 8 break down the cost of the cold and hot iteration for UW30 showing the impact of sharing on the absolute *I/O* costs in the hot and cold iterations in this workload.

We next consider recent snapshots. When the set of snapshots defined by Qs includes recent snapshots, the number of pages fetched from *Pagelog* in a given snapshot iteration is impacted by an additional factor, namely by the number of pages the snapshot shares with the current state of the database since page shared with the current state is fetched from the main memory, as explained in Section 4. Therefore, the number of pages fetched by a snapshot iteration decreases as snapshot gets closer to the end of our snapshot history and snapshots share more pages with the current state.

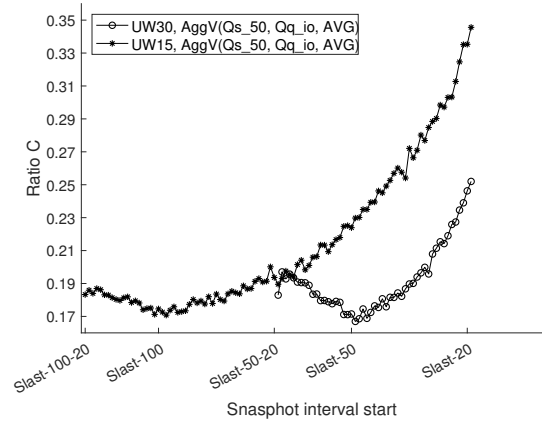


Figure 7: Ratio C with recent snapshots: impact of sharing with current state.

The ratio C , defined as ratio of measured RQL query cost to the cost of *all – cold* run, can be used to explain the additional impact of sharing pages with the current state database. Due to the database overlap, the cost of the cold iteration depends on the starting point of the interval. Therefore, the cost of *all – cold* run for two intervals with the same number snapshots drops when interval starts at a more recent snapshot. Figure 7 shows the ratio $C(x)$ for fixed size interval of consecutive snapshots (skip 1) starting at snapshot x , for intervals that include recent snapshots. We show two update workloads UW30 and UW15 exhibiting different inter-snapshot sharing $diff(S1, S2)$.

Assuming *Slast* is the most recent declared snapshot, and *OverwriteCycle* is the overwrite cycle length for a given update workload UW , the interval starting at snapshot $Slast - OverwriteCycle - 20$ is the earliest interval to include a snapshot sharing pages with the database. *OverwriteCycle* is 100 in case of UW15 and 50 in case of UW30. We consider therefore intervals starting at $x = Slast - 100 - 20$ and later for UW15, and $x = Slast - 50 - 20$ and later for UW30.

For intervals starting with an old snapshot x , $C(x)$ drops as x becomes more recent since the measured RQL cost decreases but the cost of *all – cold* run remains constant. For intervals starting with a recent snapshot x , $C(x)$ increases since the cost of *all – cold* run decreases and converges to the measured RQL cost, as both cost drop as intervals become more recent.

In absolute terms, RQL cost decreases sharply as we move to more recent intervals as shown in Figure 8, where an iteration on a more recent snapshot *Slast-25* performs significantly better than on older *Slast-50* (UW30).

Where cold iteration cost can be a dominant factor for old snapshot intervals since it can fetch substantial number of pages from *Pagelog*, this is not so for recent intervals where cold iteration fetches a substantial number of pages from the database so the dominating factor for intervals of recent snapshot is the sharing with the current state of the database.

5.2 CPU intensive queries

We expect snapshot page sharing to have less impact on CPU intensive RQL queries.

We consider two kinds of CPU intensive RQL queries. In one case, an RQL query issues a computationally heavy Qq so that SQL query execution time is the dominant cost, in the other case

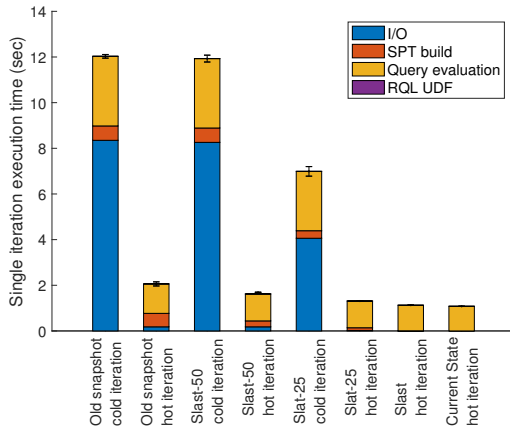


Figure 8: Single iterations cost for the RQL query `AggregateDataInVariable(Qs_50, Qq_io, T, AVG)` with update workload UW30.

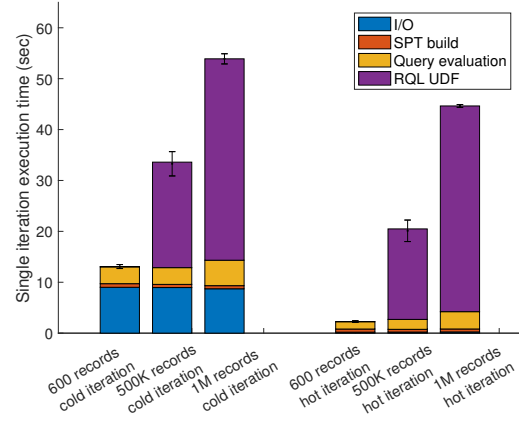


Figure 10: Single iteration cost for `CollateData(Qs_50, Qq_collate, T)` with varying Qq output size with UW30.

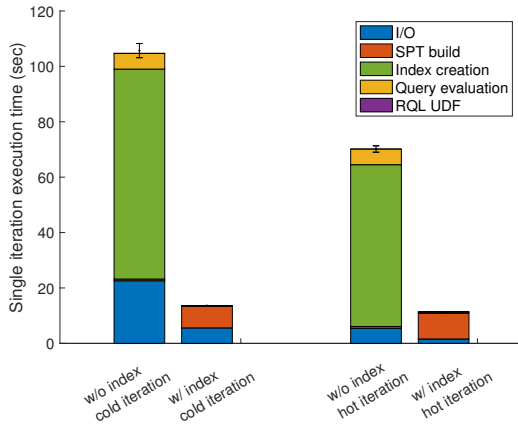


Figure 9: Single iteration cost for the RQL query `AggregateDataInVariable(Qs_50, Qq_cpu, T, AVG)` with update workload UW30.

RQL defines a Qq with a large output size. In the latter case the RQL UDF becomes the dominant cost.

For computationally intensive Qq, especially with join operations, if there is no native index in the snapshot, SQLite creates covering indices to assist the query evaluation. An experiment reported in Figure 9 shows that index creation always dominates RQL cost. Our experiment uses `AggregateDataInVariable` to avoid introducing significant RQL UDF cost and `Qq_cpu`. The Qq performs a join operation on tables `Part` and `Lineitem` and returns the revenue of the orders that include an item of a certain type. SQLite decides the building of a covering index on table `Lineitem` as part of the query execution plan. Note, unlike for I/O intensive queries, here the cost difference between a cold and hot iteration is less since I/O cost is small part of total Qq execution cost.

The Qq will not always be an *ad-hoc* query in the database workload and may have a native index built by the programmer. We evaluate the cost of the same RQL query where a native index is available in the snapshot. As shown in Figure 9 the I/O cost due

to index creation drops but instead the SPT build cost increase since an index increases the size of the database and the PageLog.

Our last experiment considers Qq that returns as a result a large number of records. This increases RQL UDF cost since SQLite UDFs invokes a callback function to perform operations for every record returned by the Qq. These operations are either insert operations in case of `Collate Data` or aggregations in case of `Aggregate Data` in Table.

Figure 10 shows the RQL performance where the CPU cost is dominated by the cost of UDF for `Collate Data` with a computationally light Qq query (`Qq_collate` in Table 1). The query scans the table `Orders` and returns the orders with order date less than a certain date. It has a single predicate which we vary to control the query output size. As in earlier CPU intensive queries, sharing has minimal impact on RQL cost.

5.3 Memory costs

The memory requirements of an RQL computation include two parts, the memory to hold the snapshot pages requested by Qq iterations, and the memory needed for result computation and to hold the result table T. Since Qq iterates over snapshots sequentially, the first part is independent of snapshot set size, and is essentially the memory needed to run Qq over a single snapshot, which includes the snapshot pages holding the working set of Qq plus the snapshot metadata structures such as `Maplog` and `SPT(S)`. The memory costs of single snapshot computation have been studied before [21]. Here we consider the memory costs of RQL mechanism result computation for different mechanisms.

RQL can support general computations over snapshots using `Collate Data` and running SQL computations over the results. However, such a method could incur high memory cost when Qs requires to compute over large number of snapshots. Memory cost can be reduced for RQL computations that perform aggregations on records across snapshots by using RQL aggregation mechanisms.

Consider the case where given the table `Orders` of TPC-H the user wants to find out, for each customer, what is maximum number of orders placed in a single snapshot by the customer and their average total price. This can be accomplished by running a single `Aggregate Data` in Table.

```
AggregateDataInTable(Qs_50, Qq_agg, T, (MAX, cn))
```

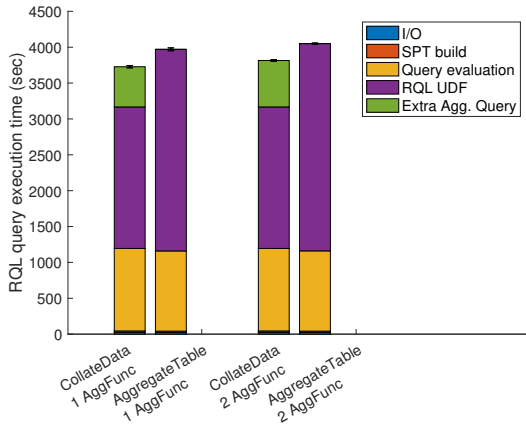


Figure 11: Comparison of RQL queries that producing the same result using Aggregate Data in Table and Collate Data for different number of aggregations.

The same result can be produced using Collate Data by first collecting how many orders each customer has placed in every snapshot in the given snapshot interval along with their average total price.

```
CollateData(Qs_50, Qq_agg, T)
```

A single SQL query can then compute the final result.

```
SELECT o_custkey, MAX(cn), av FROM T;
```

In Figure 11, the first two bars compare the performance of the two approaches for Qs that iterates over 50 snapshots. The RQL UDF cost dominates the RQL cost because the query’s output is approximately 1M of records for every snapshot. The Collate Data performs slightly better than Aggregate Data in Table.

However, Aggregate Data in Table has a significantly smaller memory footprint. Where Collate Data result table is more than 1GB, the Aggregate Data in Table result table is less than 100MB. Aggregate Data in Table achieves the memory footprint reduction for only 6% overhead in total execution time. Importantly, the memory footprint of Aggregate Data in Table is independent of Qs since we don’t expect the result table to grow significantly after each iteration, whereas Collate Data inserts the entire Qq output at each iteration.

The reason why Aggregate Data in Table is costlier than Collate Data can be explained in Figure 12 which shows single cold and hot iteration of the two RQL queries issuing the same Qq_agg. The first cold iteration is more expensive in case of Aggregate Data in Table even though they insert exactly the same number of records in the result table, because the Aggregate Data in Table creates an index on its result table. In addition, the insert operations in Collate Data is slightly cheaper than in Aggregate Data in Table because the result table of Collate Data does not have a primary key. Maintaining a primary key on a table introduces overhead to the insert operations. The reason why hot iteration is more expensive in case of Aggregate Data in Table is because we overall perform more operations. The Qq query returns approximately 1M records so, Collate Data in a single iteration executes 1M insert operations. Aggregate Data in Table on the other hand executes 1M select operation on the result table and a number of inserts or updates given the aggregation it performs. In this experiment is perform approximately 22K inserts or updates.

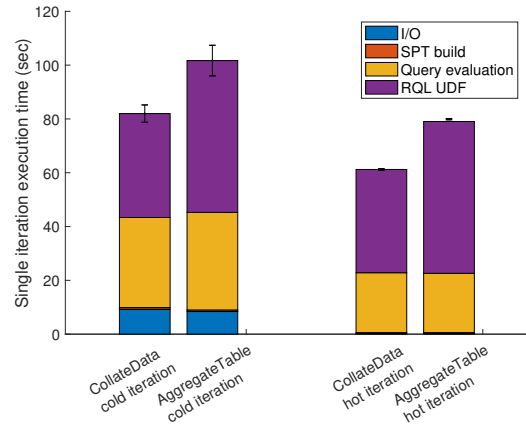


Figure 12: Single iteration cost for the RQL queries CollateData(Qs_50, Qq_agg) and AggregateDataInTable(Qs_50, Qq_agg, (MAX,cn)) and update workload UW30.

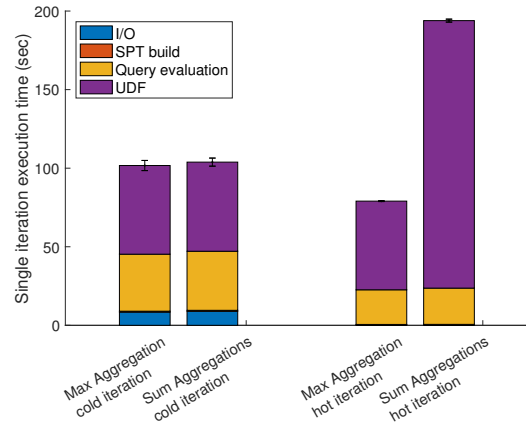


Figure 13: Single iteration cost for Aggregate Data in Table for different aggregate function

Since the Aggregate Data in Table is able to aggregate on multiple columns Figure 11 also evaluates the cost of additional aggregation. The second aggregation calculates the maximum among averages of the total price using the following Aggregate Data in Table query:

```
AggregateDataInTable(Qs_50, Qq_agg, (MAX,cn): (MAX,av))
```

And the extra query required by the Collate Date will be:

```
SELECT o_custkey, MAX(cn), MAX(av) FROM collate_result;
```

As we can see adding extra aggregation does introduce significant overhead.

For some aggregate functions like SUM and COUNT the Aggregate Data in Table have to update its result table for every record returned by the Qq. For these aggregations, we expect the hot iterations to have increased cost. Figure 13 compares Aggregate Data in Table RQL queries that apply different aggregate functions, MAX and SUM on the results of the Qq_agg. The cold iterations perform the same because they do the same initial insert operations and index creation. The hot iterations do the same number of search operations on the result table but in case of SUM they will do significantly more updates, 1M versus 22K in case of MAX.

Next, we briefly consider the Collate Data Into Intervals mechanism, comparing it to Collate Data. We focus on the memory costs since the non-memory costs of Collate Data Into Intervals closely resemble Aggregate Data In Table.

The result table size of Collate Data Into Intervals depends on the size of Q_q , and the lifetimes of records, i.e. the number of updated and deleted records between consecutive snapshots. The mechanism also needs memory to store the index. The result table size of Collate Data only depends on the size of Q_q . Our experiment uses a Q_s that defines an interval of 50 consecutive snapshots, a Q_q that returns 1.5 millions records in each snapshot (Q_q_int in Table 1), and four different update workloads UW7.5, UW15, UW30, and UW60 that respectively modify between consecutive snapshots 7,500, 15,000, 30,000, and 60,000 order records.

For Collate Data mechanism, the result table has 75M records of total size more than 3GB. For Collate Data Into Intervals, the result table has respectively 1.86M records (89 MB), 2.3M records (105MB), 2.97M records (138 MB) and 4.4M records (204 MB) for UW7.5, UW15, UW30 and UW60 update workloads. For each workload, the mechanism requires about 50% additional memory to hold the index during the computation. Interestingly, with our workloads, increasing the number of modified records between snapshots does not increase the result table size proportionally. Overall the experiment shows Collate Data Into Intervals can substantially reduce memory costs, confirming the known space saving properties of record lifetime based snapshot representation compared to naive page-level representation [24].

6 RELATED WORK

Computations over past state have been investigated in depth by a rich body of work on temporal databases [17, 27]. The most common data model adopted by the temporal databases is the extension of the relational data model with time-stamps that record lifetimes of record values via attributes indicating the start and end time [17]. The temporal dimensions can vary depending on whether a database supports the *transaction time* or the *valid time*, or both (bi-temporal databases). Temporal databases have not been widely adopted by general applications because of poor performance during normal in-production operation [10] and portability concerns.

One of the first commercial databases to support historical data management was Oracle Flashback [18] which allows to store all the modifications in a compressed format using the undo tablespace without affecting application portability. IBM provides support for bi-temporal tables in DB2 [19] by allowing the declaration of additional attributes in temporal tables to indicate the time dimension. Both Flashback and DB2 provide time travel operation to a single point in the past but offer no support for temporal computations over multiple past points which is the main focus for our work. Teradata [1] supports bi-temporal tables by extending the tables with columns representing the time domain and supports a set of temporal computations called temporal aggregations [30]. These computations scan the values which participate in the aggregation at the logical level, to determine how they have changed between different timestamps, and then compute the aggregation on the values changes.

In contrast to the logical record level approaches to past state management, RQL computations run in a snapshot system that manages the past state using a different low-level page-based approach where snapshots expose to application the entire past

state of the database, and portions of the state required by the snapshot computation are assembled on demand from the low-level representation. Using the snapshot system RQL provides multi-snapshot computations to applications in a seamless manner as we have explained. Moreover, in terms of expressive power, these computations can compute anything a record-level past state system can compute assuming snapshots capture all updates to the database. In particular, our Collate Data Into Intervals mechanism can create the same time-stamped representation used by the temporal databases. A potential downside of the snapshot system approach is that snapshot representation is less compact than logical record level representation and adds space overhead. However, prior work has shown that a snapshot system can reduce the space overhead substantially without impacting normal in-production application performance, using an adaptive low-level page-diff approach [24], that offers a convenient trade-off between more compact snapshot representation and a higher cost of snapshot reconstruction.

Temporal aggregations apply aggregate functions on relations that evolves over time, e.g. the average salary of an employee over the past certain years. Efficient methods for computing temporal aggregations have been the subject of numerous studies in temporal database research. The implementation of temporal aggregations is challenging in the logical temporal models because in order to aggregate along the time dimension a temporal computation may need to assemble the consistent state of multiple records at each time. Many of the proposed techniques in the literature accelerate this process using sophisticated data-structures that impose a variety of constraints to achieve efficiency. An early work by Kline and Snodgrass uses a non-indexed approach based on a data structure called Aggregation Tree [12]. The approach requires the entire structure to be memory resident, and has a worst case complexity of $O(n^2)$ when the tree is unbalanced. The complexity can be improved in special cases of ordered data. SB-Tree [28] and MVS-Tree [29] relax the in-memory limitation by introducing disk-based indexes for temporal aggregations. The approach limits the type of aggregate functions if data deletion is expected to SUM, COUNT and AVG. Moreover, the index size can be larger than the database. The TMDA [3] and Timeline Index [9] do not limit the aggregation functions. TMDA however does not support time travel queries. The Timeline Index is an in-memory system that requires large amount of memory to perform well. In contrast, RQL relies on state reconstruction at a lower level, and since each Retro snapshot includes the entire state of the database, RQL computations can aggregate over consistent state seamlessly without restricting types of aggregation and without requiring special temporal indexes. Nevertheless, auxiliary temporal data-structures can potentially be beneficial for repeated RQL computations and we consider this future work.

Similarly to temporal aggregations, temporal join is another challenging computation for temporal databases due to the need to identify all the versions of the join candidates that overlap in the time domain. Like with aggregation, in a snapshot system temporal join poses no addition challenge because the join candidates that overlap in time exist in the same snapshots and the temporal join is executed as if they were in current state.

Several temporal query languages have been adopted by temporal systems. The most notables are the TSQL2 [26] and TQuel [25] which extend SQL and Quel languages respectively. Some of the TSQL2 temporal properties have been adopted in SQL:2011 standard and the Teradata database. T4SQL [5] and TENORS [4] are more recent languages designed based on the time-stamping data

model. These languages are not suitable for our snapshot based system because of the different data model. Our SQL UDF based query language is fully compatible with SQL and does not require any change to the current ISO standard version. Languages designed for streaming data management such as CQL [2], resemble temporal computations since stream tuples include timestamps but since they are specialized for computations subject to real-time performance requirements they offer less expressive power compared to the temporal languages and RQL.

Although, RQL computations are most closely related to temporal databases, the snapshot system RQL extends is also similar to versioning systems with linear branching. These systems manage historical data by creating versions of the dataset somewhat similarly to the way Retro uses compact diff-based snapshot [24]. Array data versioning system [20] supports time travel queries and uses a SELECT primitive similar to ours to collect data from multiple versions. Decibel [14] is a branching system for relational datasets and supports time travel queries and multi-version aggregation. Our snapshot system does not support non-linear branching so many of the computation primitives and methods they explore are not applicable to RQL.

A snapshot system called Hyper [11] utilizes hardware-assisted memory snapshots to implement a hybrid OLTP and OLAP relational database system. OLTP and OLAP queries can run in parallel, in separate processes. The analytical queries access transaction consistent snapshots of the current state which are discarded after the queries are evaluated. Hyper does not consider a language for snapshot computations.

7 CONCLUSION

Auditing and other forms of claim checking require applications to compute over multiple past states of their data. The current systems supporting past state computations cannot be easily used by general applications using simple data stores. These applications however can easily use a snapshot system but current snapshots systems do not provide convenient support for multi-snapshot computations. Trying to bridge this gap, we proposed RQL, a retrospective query language that allows programmers to specify multi-snapshot computations in a snapshot system. Our language, implemented as a SQL extension using SQL UDF callbacks in SQLite BDB with Retro snapshot system, provides programmers a convenient way to express computations using the language of the application. Our experimental study evaluates the performance of RQL computations and explains how RQL program parameters, the SQL programs Qq and Qs interact with the page-level copy-on-write snapshot representation. This is the first study explaining the performance of programs running on multiple page-level copy-on-write snapshots. Our future work includes performance optimizations for RQL programs exploring how computations can be shared across multiple snapshots and whether parallelization can be applied.

8 ACKNOWLEDGEMENTS

This work was partially supported by National Science Foundation award CNS-1318798.

REFERENCES

[1] Mohammed Al-Kateb, Ahmad Ghazal, Alain Crolotte, Ramesh Bhashyam, Jaiprakash Chimanchode, and Sai Pavan Pakala. 2013. Temporal query processing in Teradata. In *Proceedings of the 16th International Conference on Extending Database Technology*. ACM, 573–578.

[2] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2006. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal/The International Journal on Very Large Data Bases* 15, 2 (2006), 121–142.

[3] Michael H Böhlen, Johann Gamper, and Christian S Jensen. 2006. Multi-dimensional aggregation for temporal data. In *EDBT*, Vol. 3896. Springer, 257–275.

[4] Cindy Xinmin Chen, Jiejun Kong, and Carlo Zaniolo. 2003. Design and Implementation of a Temporal Extension of SQL. In *Data Engineering, 2003. Proceedings. 19th International Conference on*. IEEE, 689–691.

[5] Carlo Combi, Angelo Montanari, and Giuseppe Pozzi. 2007. The T4Sql Temporal Query Language. In *Proceedings of the Sixteenth ACM Conference on Conference on Information and Knowledge Management (CIKM '07)*. ACM, New York, NY, USA, 193–202.

[6] Transaction Processing Performance Council. 2010. TPC-H: Decision Support Benchmark. <http://www.tpc.org/tpch>. (2010).

[7] William Endressi. 2013. On-line Analytic Processing with Oracle Database 12c. *An Oracle White Paper* (2013).

[8] Hipp, D. R., Kennedy, D. and Mistachkin, J. 2017. SQLite (Version 3.21.0) [Computer software]. (2017). <https://www.sqlite.org/>

[9] Martin Kaufmann, Peter M Fischer, Norman May, Chang Ge, Anil K Goel, and Donald Kossmann. 2015. Bi-temporal timeline index: A data structure for processing queries on bi-temporal data. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*. IEEE, 471–482.

[10] Martin Kaufmann, Peter M Fischer, Norman May, and Donald Kossmann. 2014. Benchmarking Bitemporal Database Systems: Ready for the Future or Stuck in the Past?. In *EDBT*. 738–749.

[11] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*. IEEE, 195–206.

[12] Nick Kline and Richard T Snodgrass. 1995. Computing temporal aggregates. In *Data Engineering, 1995. Proceedings of the Eleventh International Conference on*. IEEE, 222–231.

[13] David Lomet, Roger Barga, Mohamed F Mokbel, German Shegalov, Rui Wang, and Yunyue Zhu. 2005. Immortal DB: transaction time support for SQL server. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. ACM, 939–941.

[14] Michael Maddox, David Goehring, Aaron J Elmore, Samuel Madden, Aditya Parameswaran, and Amol Deshpande. 2016. Decibel: The relational dataset branching system. *Proceedings of the VLDB Endowment* 9, 9 (2016), 624–635.

[15] Olson, M. A., Bostic, K. and Seltzer, M. I. 1999. Berkeley DB. In *Proceedings of USENIX Annual Technical Conference, FREENIX Track*. Monterey, CA, USA.

[16] Fatma Özcan, Yuanyuan Tian, and Pinar Tözün. 2017. Hybrid Transactional/Analytical Processing: A Survey. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 1771–1775.

[17] Gultekin Ozsoyoglu and Richard T Snodgrass. 1995. Temporal and real-time databases: A survey. *IEEE Transactions on Knowledge and Data Engineering* 7, 4 (1995), 513–532.

[18] Ravi Rajamani. 2007. Oracle total recall/flashback data archive. *An Oracle White Paper* 12 (2007).

[19] Cynthia M Saracco, Matthias Nicola, and Lenisha Gandhi. 2010. A matter of time: Temporal data management in DB2 for z. *IBM Corporation, New York* (2010).

[20] Adam Seering, Philippe Cudre-Mauroux, Samuel Madden, and Michael Stonebraker. 2012. Efficient versioning for scientific array databases. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*. IEEE, 1013–1024.

[21] Ross Shaull. 2013. *Retro: a methodology for retrospection everywhere*. Brandeis University.

[22] Ross Shaull, Liuba Shrira, and Barbara Liskov. 2014. A Modular and Efficient Past State System for Berkeley DB.. In *USENIX Annual Technical Conference*. 157–168.

[23] Ross Shaull, Liuba Shrira, and Hao Xu. 2008. Skippy: a new snapshot indexing method for time travel in the storage manager. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM, 637–648.

[24] Liuba Shrira and Hao Xu. 2006. Thresher: An Efficient Storage Manager for Copy-on-write Snapshots.. In *USENIX Annual Technical Conference, General Track*. 57–70.

[25] Richard Snodgrass. 1987. The temporal query language TQuel. *ACM Transactions on Database Systems (TODS)* 12, 2 (1987), 247–298.

[26] Richard T. Snodgrass (Ed.). 1995. *The TSQL2 Temporal Query Language*. Kluwer.

[27] Abdullah Uz Tansel, James Clifford, Shashi Gadia, Sushil Jajodia, Arie Segev, and Richard Snodgrass. 1993. *Temporal databases: theory, design, and implementation*. Benjamin-Cummings Publishing Co., Inc.

[28] Jun Yang and Jennifer Widom. 2001. Incremental computation and maintenance of temporal aggregates. In *Data Engineering, 2001. Proceedings. 17th International Conference on*. IEEE, 51–60.

[29] Donghui Zhang, Alexander Markowetz, Vassilis J. Tsotras, Dimitrios Gunopoulos, and Bernhard Seeger. 2001. Efficient Computation of Temporal Aggregates with Range Predicates. In *Proceedings of the Twentieth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 21-23, 2001, Santa Barbara, California, USA*.

[30] Xin Zhou. 2011. Processing a temporal aggregate query in a database system. (Aug. 30 2011). US Patent 8,010,554.

Big Data Analytics for Time Critical Maritime and Aerial Mobility Forecasting

George A. Vouros⁺,
Christos Doulkeridis,
Georgios Santipantakis,
Akrivi Vlachou,
Nikos Pelekis,
Harilaos Georgiou,
Yiannis Theodoridis,
Kostas Patroumpas
University of Piraeus, Greece
⁺Corresp. Author email:
georgev@unipi.gr

Elias Alevizos,¹
Alexander Artikis^{1,2}
¹IIT, NCSR 'Demokritos',
²University of Piraeus, Greece
Georg Fuchs, Michael Mock,
Gennady Andrienko,
Natalia Andrienko
Fraunhofer Institute IAIS,
Sankt Augustin, Germany

Cyril Ray,
Christophe Claramunt
Ecole Navale / ENSAM, France
Elena Camossi,
Anne-Laure Joussemme
CMRE, La Spezia, Italy
David Scarlatti
BR&T-E, Spain
Jose Manuel Cordero
CRIDA, Spain

ABSTRACT

The correlated exploitation of heterogeneous data sources offering very large archival and streaming data is important to increase the accuracy of computations when analysing and predicting future states of moving entities. Aiming to significantly advance the capacities of systems to improve safety and effectiveness of critical operations involving a large number of moving entities in large geographical areas, this paper describes progress achieved towards time critical big data analytics solutions to user-defined challenges in the air-traffic management and maritime domains. Besides, this paper presents further research challenges concerning data integration and management, predictive analytics for trajectory and events forecasting, and visual analytics.

1 INTRODUCTION

Aerial and maritime transportation have significant role and impact on the global economy and our everyday lives. The improvements along the last decades of these transportation means in terms of management, planning, security, information to operators and end-users has been driven by location-based information. The ever-increasing volume of data emphasizes the need for advanced methods supporting real-time detection and prediction of events and trajectories, together with advanced visual analytic methods, over multiple heterogeneous, voluminous, fluctuating, and noisy data streams of moving entities.

These transportation domains aim at fostering collaborative decision-making environments, involving all the stakeholders in the process as this is expected to provide direct and positive consequences in terms safety, efficiency and economy in both, aerial and maritime domains. For instance, by having a better understanding of the air navigation data (historical data of flight plans, sector configurations and weather), the number of published regulations (e.g. delays imposed to flights entering congested areas), which limit the number of flights planned to enter an airspace or aerodrome to match traffic demand to available capacity, could be more accurately forecasted and thus the adherence to scheduled trajectories improved, reducing delays and operational costs.

Tracking and analysis of vessels' behaviour at sea are also important challenges for enhancing the safety and efficiency of many maritime operations [10]: preventing ship accidents by monitoring vessels' activity represents substantial financial savings for shipping companies (e.g., oil spill clean-up) and averts irrevocable damages to maritime ecosystems (e.g., fishery closure).

The current Air Traffic Management (ATM) is nowadays changing its point of view from a time-based operations concept to a Trajectory-Based Operations (TBO) one, which means a better exchange, maintenance and use of the aircraft trajectories. Similarly, real-time tracking and forecasting of trajectories of ships from port-to-port, worldwide, together with route prediction and early recognition of maritime events, are essential to improve safety of operations at sea. More accurate and richer information on trajectories and related events is expected to increase the abilities to predict trajectories and forecast events, anticipate the behaviour of any moving entity, improve situational awareness, and consequently the decision-making process in both ATM and maritime domains.

Due to the complexity of these transportation systems, as well as due to factors contributing to increased uncertainty and lack of accuracy in the mobility data, the current techniques for predicting trajectories are limited to a short-term horizon, while the event detection and forecasting abilities are limited. The development of methodologies exploiting the amount of data from heterogeneous data sources, managing the possible lack of veracity for (actual, historical and planned) trajectories and other contextual aspects (e.g. airspace sector configurations, regulations and policies, sea protected areas, weather patterns), is expected to overcome some of the limitations of existing systems.

The objective of this paper is to describe progress achieved towards big data analytics solutions concerning moving entities in the air-traffic management and maritime domains, and to present related research challenges on data integration and management, predictive analytics for trajectory and events forecasting, and visual analytics. Challenges, methods and techniques developed in this paper originate from the datAcron project (<http://www.datacron-project.eu/>) whose aim is to provide advances for Big Data Analytics for Time Critical Mobility Forecasting.

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

The rest of the paper is organised as follows. Section 2 introduces the challenges from both domains from an end-user perspective. Section 3 presents the overall datAcron architecture. Section 4 presents the data management components and the datAcron ontology, while Section 5 presents the location and trajectory predictors. Section 6 presents the events detection and forecasting components, and Section 7 the online and offline visual analytics components. All sections present experimental results, providing evidence of the progress achieved towards time critical (i.e. real time) data processing and mobility analytics tasks. Finally, Section 8 draws the conclusions.

2 USER-DEFINED CHALLENGES IN THE ATM AND MARITIME DOMAINS

Efficiency in the air-traffic management system requires minimizing costs for both the airspace users (mainly airlines) and the operators (namely, Air Navigation Service Providers, ANSP's). In general, one key enabler for reducing costs is the predictability of the system. In particular, from the point of view of the ANSP, maintaining the balance between the demand (i.e., the number of users trying to use limited resources like airports, airspace sectors, etc.) and the capacity (i.e., the number of users which can safely use the mentioned resources) is one of the main challenges. For an airline, flying according to the plan, avoiding delays or extra fuel consumption represents the ideal to achieve daily operations. A trajectory based operation approach enables to plan which resources of the air-traffic management system will be used by each flight (airports, airways, sectors, etc.), define the achievable schedules, as well as the implied costs, increasing efficiency.

Big data technology, which can exploit very large historical and streaming data sources for positioning, contextual aspects and weather, presents opportunities to boost current predictability capacities that are based mainly on complex theoretical models of the different components of the air-traffic management system.

Surveillance is an ever-increasing data source since new technologies are deployed (like ADS-B) which allow to collect data more widely (space based ADS-B promises global coverage) and more frequently. Weather data, identically, each time is offered with more resolution, both geographical and temporal. Contextual data, like flight plans, waypoints, or airways is increasing, linked to the traffic growth, year after year. While each data set is big, correlating and jointly exploiting all of them together is what makes big data technology necessary. The aircraft trajectory must be understood not only as the 4D collection of points: It should also include events relevant for the traffic management and the airline operations. So, predicting the aircraft trajectory implies predicting these events too, and vice versa. The amount of information involved in this trajectory prediction process requires advanced visual analytics aids in order to understand the patterns of the predicted trajectories and events, inspect the exact reasons for deviating from plans towards either making adjustments to the actual system, or tune trajectory and event detection and prediction methods for more accurate results.

Accurate predictions of trajectories will further advance adherence to flight plans (i.e., intended trajectories) reducing many factors of uncertainty, allowing stakeholders to do better planning of the operations, reducing risk of disruptions. Our maritime scenarios [13] aim to address operational concerns regarding fishing activities, highlighting the need for continuous, timely (i.e.

real time) tracking of fishing vessels and surrounding traffic, as well as the need for offline data analytics.

Security in fishing addresses the need to detect and foresee collisions between ships, potentially optimizing rendezvous between rescuing ships in proximity of a vessel in danger and emergency services. Collision avoidance is a typical situation to be addressed: To prevent collision of fishing vessels with other ships we need to predict which other vessels (such as cargos, tankers, ferries) will cross the areas where the fishing vessels are fishing, sending a warning to the vessels identified for possible collision, taking also appropriate action as specified by COLREGs¹. To advance decision making in these cases the potential risk assessment should be as accurate as possible. Such a development could also be used on board to enhance situational awareness, when it is anticipated that a vessel will be required to "give way" to a fishing vessel.

Additionally, we need to detect vessels in distress, and further detect vessels in their vicinity to optimise rescuing operations. Analytics for detecting fishing patterns that are robust to noise and lack of veracity in data, as well as accurate trajectory prediction algorithms, are fundamental to support effectively those operational requirements.

Sustainable development maritime scenarios supporting the monitoring of fishing activities' impact, including the illegal ones, is of immense importance. In particular, towards the protection of areas from fishing we address the issue of Illegal Unreported Unregulated (IUU) fishing, which is a global threat to the preservation of maritime ecosystems and could potentially undermine the sustainable development in large areas of the world that depend on maritime resources. Beside the introduction of maritime protected areas where protected species live and where navigation is prohibited, fishing seasons are regulated and fishing activities are forbidden in certain periods of the year, depending on the area and on the type of catch. Towards these objectives we need to predict and detect vessels entering, exiting, sailing, spending time or fishing in geographical zones.

Given the above cases in both domains, real-time integration of disparate data sources enabling scalability for massive amounts of dynamic data is an existing challenge, which is very closely connected to the maritime domain, as well as to the ATM domain. Table 1 presents the main data sources exploited in datAcron².

A series of specific challenges concerning processing and managing data from these sources are as follows: (a) scalable, automatic, real-time processing, semantic annotation and linking of data towards coherent views on integrated cross-streaming (data-in-motion) and archival (data-at-rest) data; (b) incremental integration of data, allowing advanced management and query-answering of spatio-temporal data; (c) efficient distributed management and querying of integrated spatio-temporal data.

Revolving around the notion of trajectories and further making advances towards trajectories and events' detection and prediction, both domains present the following challenges: (a) real-time reconstruction of entities' trajectories, supported by real-time processing and analysis of streams of data; (b) algorithms for the prediction of anticipated trajectories at different time scale; (c) algorithms for complex event recognition and prediction in real-time.

¹www.imo.org/en/About/Conventions/ListOfConventions/Pages/COLREG.aspx

²Vessels equipped with the Automatic Identification system (AIS) communicate their positions continuously. Coastal stations and satellites receive messages in real-time.

Table 1: The datAcron surveillance weather and contextual data sources (spatial coverage is Europe).

	Type	Source	Format	Volume	Velocity
Surveillance	Maritime	Terrestrial AIS	CSV Files	19,680,743 messages (1.05 GB) for 6 month	~ 76 messages per min (in average)
		Mixed terrestrial and satellite AIS	CSVFiles	81,722,110 messages (8.11 GB) for one month	~ 1,830 messages per min (in average)
		Mixed terrestrial and satellite AIS	Stream of messages in JSON	~ 400 KB / min (in average)	~ 3,700 messages per min (in average)
	ATM	FlightAware	Stream of messages in JSON	13GB/day	1.2Mb/s
		IFS Radar Tracks	CSV Files	12GB/day (Spanish Airspace)	1.1Mb/s
Weather	Maritime & ATM	Sea state	CSVFiles	79,652,684 forecasts (3.02 GB)	1,463 forecast files - 1 file / 3 hours
		Weather forecast	CSVFiles	71,516 observations (5 MB)	1 obs/hour, from 16 stations
Contextual	Maritime	Geographical	ESRI shapefiles	22 different features (1.4 GB)	Static
		Port Registers	ESRI shapefiles	5,754 different ports (70 MB)	Static
		Vessel Registers	CSVFiles	166,683 distinct ships	Static
	ATM	Eurocontrol NM B2B	CSV Files	1.7 GB/day	Static
		Eurocontrol NM B2B	Flat Files	30MB/cycle	Static
Other (ATM)	Eurocontrol	CSV Files	30MB/month	Static	

Nevertheless, Visual Analytics (VA) [33] creates opportunities for a synergy between human analysts and computers by providing appropriate visual interfaces to all facets of analytical reasoning, from data exploration, pattern discovery and outlier identification, to prediction validation. It therefore facilitates the inclusion of the human domain expert’s tacit knowledge and his capabilities for reasoning and intuition into the decision process, which are of fundamental importance in surveillance activities. The most important VA research challenges for both domains are as follows: (a) interactive pattern extraction considering archival and streaming data, supporting the validation of early alerts obtained by the analysis tools; (b) building situation overview and situation monitoring, capable of providing the overall operational picture of mobility at desired scales and levels of detail, both in spatial and temporal dimensions.

3 THE DATAACRON SYSTEM ARCHITECTURE FOR TIME CRITICAL MOBILITY FORECASTING

Critical mobility operations require integrating data that stems from a wide variety of diverse data sources, both archival and streaming, having all big data characteristics. During data acquisition, various tasks need to be performed, including data cleaning, compression, transformation to a common representation model, and data integration. Besides real-time operations that must be supported with minimum latency requirements, there exists a need for offline analysis of the integrated data in order to discover patterns and extract useful knowledge.

To address these challenging requirements, the datAcron system architecture, depicted in Figure 1, has been devised as a Big Data architecture for processing both real-time and archival data. While it bears similarities with the Lambda architecture [19], since it encompasses both a real-time and a batch processing layer, these layers for different purposes (e.g. online trajectory/events forecasting vs offline trajectory clustering and visual analytics over archival data).

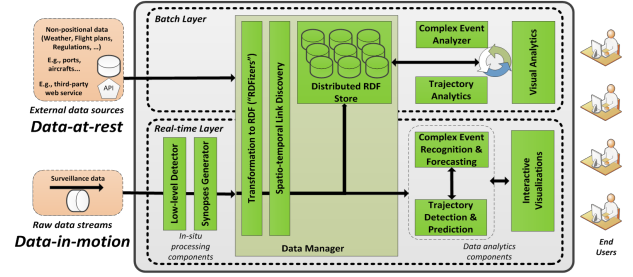


Figure 1: The datAcron system architecture.

In the real-time layer, streaming surveillance data describing the positions of moving entities, collected from terrestrial and satellite receivers are fed into the system, while several operations are performed: Statistics (min/max/avg) are computed over properties, such as speed and acceleration, in an online fashion; online data cleaning of erroneous data, as well as trajectory reconstruction and compression are performed. The goal is to provide only the data that is needed for analytics tasks: This is mainly done by generating synopses of trajectories, which are annotated towards the construction of “meaningful” trajectories. Thus, the generated trajectory synopses are transformed to RDF (Resource Description Framework) form, according to the datAcron ontology, thereby facilitating the expression of links with data originating from other sources. To this end, spatio-temporal link discovery is performed resulting in semantically enriched trajectories. Further online analysis of enriched trajectories aims at: (a) deriving predictions of the future location of a moving object, and (b) complex event recognition and forecasting. Finally, real-time visualizations support human interaction with the datAcron system.

In the batch layer, the enriched trajectories as well as data from other sources transformed in RDF are collected for persistent storage, in order to support offline data analytics. Due to the immense data volume, parallel data processing is performed over RDF data stored in a distributed way. On top of the distributed RDF store, higher-level data analysis tasks run, in order to perform trajectory analysis (clustering, sequential pattern mining) and towards building models for complex event recognition and forecasting using machine learning techniques. Last, but not least, visual analytics provide the ability to discover hidden knowledge and patterns, by means of interaction with a domain expert or a data analyst, further improving situation awareness, as well.

Below, we describe the main components of the architecture.

In-situ processing components. In-situ processing allows computation as close to the sources as possible, thus reducing communication and latency. In datAcron, we apply in-situ processing on the streaming surveillance data, as it is ingested in the system. This supports computing statistical measures of moving entities’ properties (such as speed and acceleration) and executing low-level event detection, annotating positions of moving entities with information regarding entry/exit to/from geographical areas of interest. In addition to that, trajectory compression aims to retain only a small set of positions of moving entities, also called *critical points*, without sacrificing the accuracy of the representation significantly.

Data manager. The data manager is responsible for providing a common representation of all data sources by integrating and linking data in a knowledge graph, and for query processing over that graph. To support linking of data from different

sources in a common representation format, we opt for RDF. First, any incoming data (no matter whether streaming or archival) is lifted to RDF, by means of RDF generators. The obtained representation is based on the datAcron ontology [28] (also in: http://ai-group.ds.unipi.gr/datacron_ontology) supporting the exploitation of semantically enriched information. Data interlinking is achieved via a spatio-temporal link discovery framework, which is designed to operate on streaming data sources, apart from archival. Finally, the integrated spatio-temporal RDF data is stored in a distributed way, supporting spatio-temporal RDF query answering by means of a parallel processing engine for RDF data, offering batch processing and analysis, with notable difference to existing solutions (see [1] for a recent survey).

Trajectory detection and prediction. This component predicts the future location of moving entities in real-time, exploiting enriched trajectories offered by the data manager. The trajectory prediction component complements the future location predictor, while offline trajectory analytics (not in the scope of this article) over distributed RDF data are delivered by the corresponding component.

Complex event recognition and forecasting. This component targets the need to detect and forecast complex events related to the movement of moving entities. To detect and forecast events in a timely fashion, a novel technique using Pattern Markov Chains is proposed for continuous narrative assimilation on data streams. In addition to that, machine learning methods are applied to build prediction models, while an offline *complex event analyser* operates on the historical data and discovers patterns of events to be predicted. The latter are not within the scope of this article.

Visual analytics. The aim is to support exploratory and interactive analysis of data, in order to enable the task of human interpretation, which is necessary in the case of Big Data. Visual analytics does not represent a single, specific analysis technique but rather a methodological approach to gain insight into large, complex, noisy and often conflicting data, to develop and test hypotheses, and to build and understand complex analytical models. The key aspect is the collaborative work between the computer and the human analyst, whereby the human expert imparts background knowledge about the current analysis task's context and reasoning in the overall analytical process.

For the implementation of the overall architecture, the big data technologies employed include a blend of state-of-the-art solutions that are used in production environments successfully. Stream processing components have been developed in Apache Flink, harnessing the scalability and low latency offered. Solutions that rely on micro-batching, such as Spark Streaming, do not match with the required complex stream processing and low latency requirements targeted by our work. Instead, for batch processing and analysis, we have selected Apache Spark which is a more mature project than Flink, having a larger ecosystem, and bigger user base, while also achieving scalability, high performance, and exploiting in-memory processing. The stream-based communication between components is achieved by means of Apache Kafka.

4 DATA PROCESSING AND MANAGEMENT

Aiming to build solutions towards managing data that are connected via, and contribute to enriched views of trajectories upon which ATM and maritime challenges focus, we revisit the notion of semantic trajectory and built on it towards integrating the

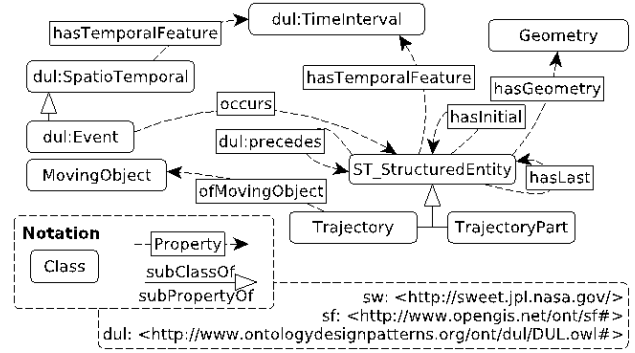


Figure 2: The main concepts and relations of the ontology.

wealth of information available in heterogeneous data sources in both domains in a representation where trajectories are the main entities: The datAcron ontology (http://ai-group.ds.unipi.gr/datacron_ontology) has been designed to provide a common model for all data sources in both domains towards supporting analysis tasks. Its development has been driven by ontologies related to our objectives (e.g. DUL, SimpleFeature, NASA Sweet and SSN) as well as schemas and specifications regarding data sources from both domains. To a greater extent than other models for representing trajectories, this ontology provides the means for specifying trajectories at varying levels of spatio-temporal analysis: Trajectories can be seen as temporal sequences of moving entities' positions derived from raw data, as raw data aggregations signifying meaningful events providing a synoptic view of raw trajectories (generalizing on the stops and moves model [31], according to the types of critical points), as temporal sequences of meaningful trajectory segments (each revealing specific behaviour, event, goal, activity etc.), or as mere geometries. Representations at any such level of analysis are linked to each other, as well as to related information and events. Beyond answering spatio-temporal SPARQL queries concerning trajectories along with information regarding aspects that affect and are affected by the mobility of moving entities, this ontology supports generic data transformations for adapting available data to the analysis goals, or to specific requirements of analysis tasks. This is done by converting movement data from one form to another, to support different task foci: movers, spatial, events, space, and time. Details are provided in [28].

According to the ontology specifications, as illustrated in Figure 2, a Trajectory can be segmented to TrajectoryParts, each including other segments and/or semantic nodes. Each semantic node may be associated with a specific raw position or a temporally ordered sequence of raw positions of a moving object. Trajectories and trajectory parts can be associated with any relevant information, as well as with events (*dul:Event*). Although events may happen independently from the trajectory, we focus on those happening on the trajectory itself (e.g. a "turn" or a "gap of communication") and on those concerning moving entity state (e.g. vessel in a protected or in a bad-weather area). The detailed patterns for specifying structured trajectories and occurring events are presented in [28].

4.1 Data processing

The *low-level event detection* component is aiming at enriching the raw-data generated by the moving entities with basic derived attributes that serve as input for higher-level processing. A major

consideration is to achieve that enrichment with low-latency, preferably by processing streaming data close to data source (in-situ): This provides a number of inherent advantages, such as decreased communication delays, savings in communication, and reduced overhead in sub-sequent evaluation steps.

The detection of low-level events refers to generating metadata on incoming raw data for detection of erroneous data, ensuring data quality, and enriching the data stream with contextual information for further analysis. For supporting the data quality assessment, described in Section 7, attributes of min/max, median/average of properties (e.g. speed, acceleration etc.) are generated on a per trajectory basis. In addition to that, raw position data are enriched with low-level events of entering or leaving of moving entities from one area to another one, by processing the real-time stream of moving entity positions.

The *Synopses Generation* detects important *mobility events* along trajectories represented as *critical points*: This task has to be carried out in a timely fashion against the streaming positional updates received from a large number of moving entities. Instead of retaining every incoming position for each object, the *Synopses Generator* module drops any predictable positions along trajectory segments of “normal” motion characteristics, since most vessels and aircrafts usually follow almost straight, predictable routes at open sea and in the air, respectively. By doing so we may only retain positions that signify changes in actual motion patterns. We opt to avoid costly trajectory simplification algorithms like [18, 21] operating in batch fashion, online techniques employing sliding windows [20] or safe area bounds for choosing samples [21], as well as more complex, error-bounded methods [16, 17]. Instead, emanating from the novel trajectory summarization framework introduced in [25, 27], specifically for online maritime surveillance, but significantly enhanced with additional noise filters and also extended for the needs of the aviation domain, the *Synopses Generator* applies single-pass heuristics for achieving succinct, lightweight representation of trajectories. We prescribe that each trajectory can be approximately reconstructed from judiciously chosen critical points of the following types:

- *Stop* indicates that an entity remains stationary (i.e., not moving) by checking whether its instantaneous speed is lower than a threshold over a period of time.
- *Slow motion* means that an entity consistently moves at low speed over a period of time (below a given threshold).
- *Change in Heading*: Once there is an angle difference in heading greater than a given threshold with respect to the mean velocity vector (computed over the most recent course), the current location should be emitted as critical.
- *Speed change*: Such critical points are issued once the rate of change for speed exceeds a given threshold with respect to its mean speed over a recent time interval.
- *Communication gaps* occur when an entity has not emitted a message over a time period, e.g., the past 10 minutes.
- *Change in Altitude* may be detected for aircrafts by checking their rate of climb (or descent), i.e., the vertical speed of the aircraft (in feet/sec) when ascending (respectively, descending). Once, this value exceeds a given threshold, a critical point should be issued in the synopsis.
- *Takeoff* is the latest position of an aircraft while on the ground: The next position has altitude above ground.
- *Landing* for flying aircrafts is the first reported location when they touch the ground.

This module can achieve dramatic compression over the raw streaming data with tolerable error in the resulting approximation. At lower or moderate input arrival rates, data reduction is quite large (around 80% with respect to the input data volume), but in case of very frequent position reports, compression ratio can even reach 99% without harming the quality of the derived trajectory synopses (typically, straight movements with constant speed).

Empirical results [25] indicate that such critical points can be emitted in real-time keeping in pace with the incoming raw streaming data. As a next step, we plan to address the case of cross-stream processing, i.e., correlating surveillance data from multiple sources in order to provide a coherent trajectory representation.

4.2 Data management

To convert the data from different sources into the common RDF model and integrate them in a knowledge graph, we designed and implemented a generic RDF generation framework, which can be instantiated to any of the given (streaming or archival) data sources. Due to the syntactic and semantic heterogeneity of data sources exploited in datAcron, and given that sizes vary from a few thousands (e.g. aircraft or vessel registries), to practically infinite streams of data (e.g. reported positions of moving entities), we need an efficient method that can easily be integrated to widely used SPARQL workflows, to rule all the data sources, and that will also be easily adapted to changes on both the ontology and the sources, while the output will be easily verified. The proposed method stands on two main components: (a) The *data connector*, which is responsible to connect to a data source and accept the data provided. It is capable of applying basic data cleaning operations, computing and converting values, applying simple filters, or extracting information regarding the incoming entries, e.g. extracting the Well-Known-Text representation of a given geometry in a Shapefile. (b) The *triple generator*, which is responsible to convert all the data coming through the data connector, into meaningful triples w.r.t. the datAcron ontology. Triple generators exploit graph templates and variables vectors. The variables vectors enable transparent reference to datasource fields, while they enable the RDF generation method to refer to data not explicitly available in the source, but generated during the generation process. The graph template, on the other hand, uses these variables into triple patterns; i.e. in triples where any of the subject or object can be either a variable or a function with variables as arguments. Such an example of data conversion into triples by exploiting a graph template made of triple patterns, is provided in Figure 3.

By doing so, and in contrast to other RDF generators, the proposed method needs no further knowledge of a specific vocabulary (e.g. compared to RML [11]), and it can be used by anyone who can write simple SPARQL queries. It requires no underlying SPARQL engine, and it inherently supports parallelisation and streaming data sources (compared to SPARQL-Generate [15] and GeoTriples [14]).

This RDF generation method manages to transform 10,500 input records to RDF per second. For some sources, this number may be smaller due to complicated geometries. Overall, the average time per triple generated is approximately 0.04 seconds, given that the frequency of position reporting per aircraft/vessel is at least 2 seconds.

Triples produced by the RDF generators are directed to the *datAcron link discovery* component: This detects spatio-temporal and proximity relations such as “within” and “nearby” relations between stationary and/or moving entities. It is noteworthy that there is not much work on the challenging topic of spatio-temporal link discovery nor on link discovery over streaming data sets. State of the art approaches such as [23], [30], [29] focus on spatial relations in static archival data sets only. In particular RADON [29] employs optimizations that can be only applied if the data sets are a-priori accessible as a whole, which cannot be assumed for streaming data sets. Our work addresses explicitly proximity and spatio-temporal relations in both archival and streaming datasources.

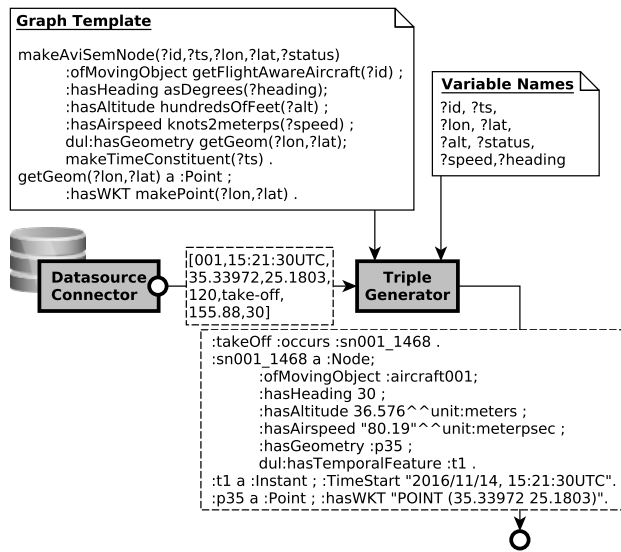


Figure 3: Triple Generation example.

The implemented component continuously applies SPARQL queries on each RDF graph fragment produced by an RDF generator, to filter only those triples relevant to the computation of a relation r . It applies a blocking method to organize entities (either being moving or stationary entities), and a refinement function to evaluate pairs of entities in any block.

Aiming to discover spatio-temporal relations among entities, existing methods use an equi-grid which organizes entities by space partitioning. The temporal dimension is not partitioned: given a temporal distance threshold, we can safely clean up data that are out of temporal scope, i.e. entities that will never satisfy the temporal constraints of the relations. To effectively prune candidate pairs of entities, the proposed method computes the *mask* of cell: This is the complement of the union of those spatial areas that correspond to entities in a cell and intersect with the cell area. Figure 4 depicts examples, where the green regions illustrate the mask of cells generated from 8,599 Natura2000 and fishing regions around Europe.

Thus, for each new entity we identify the enclosing cell, and then we evaluate that entity against the spatial mask of the cell. If it is found to be in the mask, we do not need to further evaluate any candidate pair with entities in that cell. In addition to masks, the link discovery component uses a book-keeping process for cleaning the grid, towards identifying proximity relations among entities when dealing with streamed data.

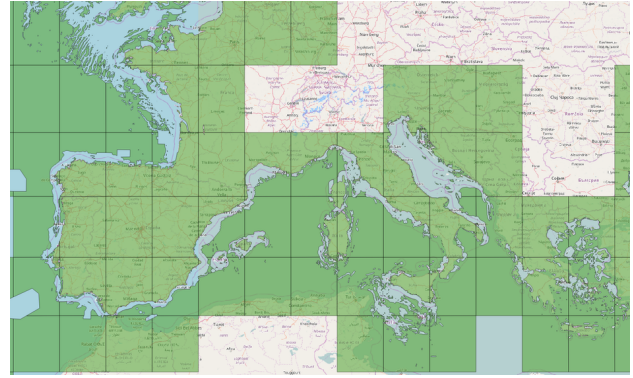


Figure 4: Equi-grid with masks for stationary areas.

We have evaluated the performance of the Link Discovery method with and without cell masks on a data set of 4,765,647 critical points, against a data set of 8,599 regions generating 381,262 `dul:within` and 9,122 `geosparql:nearTo` relations. The method without masks achieves linking 23.09 entities per second, while activation of the mask boosts the throughput to 123.51 entities per second. Preliminary results concerning `geosparql:nearTo` relations among critical points, as well as critical points and 3,865 ports, have shown a throughput of 328.53 entities per second, producing 2,536,967 relations. Challenges lying ahead for link discovery, include both, the identification of more complex spatio-temporal relations in real-time streaming data, and improving performance and scalability. The latter can be achieved by the refinement of blocking schemes for achieving better load-balancing for tasks’ parallelization, as well as by the use of advanced techniques for reducing the number of comparisons in the cells.

As far as the *Knowledge Graph Store* is concerned, even though there exist several solutions for distributed RDF processing (see [1] for a survey), a notable difference is that we deal with mobility data that have a strong spatio-temporal flavour and typical queries also contain spatio-temporal constraints. A typical distributed RDF processing engine cannot process efficiently spatio-temporal constraints, as such constraints would have to be enforced in a post-processing step to obtain the final result, at the cost of having computed a much larger set of candidate results. Motivated by this limitation, we designed a solution for scalable processing of spatio-temporal RDF data. The system contains a distributed storage layer, and a batch processing layer developed in Apache Spark. In the storage layer, we use a custom dictionary encoding technique for representing spatio-temporal entities. Our encoding technique allows representing an approximation of the position of any moving entity using a unique integer identifier, which corresponds to the spatio-temporal cell where the entity is located. We support different storage layouts, including “one-triples-table”, vertical partitioning, and property tables. Also, for the file layout we exploit Parquet, which provides a columnar layout and achieves compression. The RDF triples are stored in HDFS, while the dictionary needs to be stored in main memory for efficient access, so we opt for REDIS which fits our needs, although other NoSQL key-value stores may also comprise solutions (e.g., Aerospike, etc.) In the processing layer, we have developed different implementations of basic operators (such as filtering and join) that can be used to generate different physical execution plans from a given logical plan. Moreover, the spatio-temporal encoding is used during query processing,

by filtering triples that do not match with the spatio-temporal query constraints. This happens in parallel to filtering RDF triples, matching with the RDF graph patterns specified in the query [24].

Experimental results [24] performed over more than 269M RDF triples from surveillance, weather, and contextual data sources show that we can improve query processing time for star join queries with spatio-temporal constraints when using our techniques.

5 TRAJECTORY PREDICTION

The prediction of a trajectory evolution can be seen either (a) as a Future Location Prediction (FLP), or (b) as a Trajectory Prediction (TP) problem. In FLP, the task is to predict the next k points in the trajectory, a process that is inherently dynamic and continuously adaptive, exploiting measured (reactive mode) or predicted (proactive mode) error as feedback. On the other hand, TP aims to produce a “best guess” of the complete trajectory in the maximum likelihood sense. The two tasks are interconnected and applied in parallel, with FLP (TP) being the short-term online (full-length offline, respectively) predictor. Generally, there are two main approaches in addressing these prediction tasks:

(a) The *Kinetic approach*, which describes the forces and momentums that describe the motion of the moving entity in terms of physical laws. The kinetic approach can produce accurate predictions, but requires high-intensity processing, due to detailed simulation. Moreover, predictions, being sensitive to changes in many of the (stochastic) parameters involved, are quickly deviating, as the temporal window expands. In the aviation domain, the kinetic approach uses extremely accurate aircraft performance models, such as BADA (Base Of Aircraft Data), combined with localized weather forecasts. Similar kinetic approaches are used in various forms, e.g. for dead reckoning in navigation modules, in the maritime domain.

(b) The *Kinematic approach*, which considers only the temporal evolution of the model’s parameters as time series and exploits the causalities discovered. In practice, this includes data-driven methods that exploit enriched trajectories as training sets for FLP and TP purposes. In other words, the model “learns” the kinetic behaviour of the moving entity by processing historical information of its own trajectory in case of FLP or of an entire group of “similar” trajectories in case of TP.

In contrast, the data-driven FLP and TP targeted in datAcron rely exclusively on reference points of actual trajectories, enriched with features (e.g. weather conditions, operational constraints, etc.) that affect trajectories.

The current state-of-the-art in data-driven approaches for FLP and TP range from standard signal processing to advanced regression learners. In FLP, standard regression methods, as well as motion-type modelling have been applied primarily in the short-term time frame [32]. Since TP includes complete trajectories, a number of supervised and unsupervised methods have been applied in the context of classification: Grouping together “similar” trajectories and predicting new ones based on these groupings for “similar” input conditions, e.g. for the same departure/destination, same weather conditions, etc. [34]. The current state-of-the-art approaches do not address the range of options from short to long term predictions in its entirety, nor exploit the full enrichment of the data points as constraints to optimize the training. Additionally, the volume and velocity of the data are considered of less or even no importance compared to the spatio-temporal prediction accuracy.

Recursive Motion Functions (RMF) for FLP: Mobility patterns over short-term time frames are often studied in the sense of online predictive analytics, i.e., involving small set of positions as “recent history” and strict constraints with regard to storage and processing resources. Tao et al. [32] propose Spatio-Temporal Prediction trees as an indexing scheme supporting predictive queries and incorporating a general framework that computes different *non-linear motion patterns* to capture movements of arbitrary characteristics. In this context, the Recursive Motion Function (RMF) approach enables the computation of different types of movement (such as linear, polynomial, circular, etc) by exploiting the recent past of an object’s position sequence and adapting the prediction model according to its specific characteristics. According to our knowledge RMF is the most prominent candidate for addressing the online FLP task and under big data specifications. RMF captures the motion dynamics of an entity in a differential recursive formula by combining the most recent data points per f (system parameter) and is most effective when the acceleration components are zero, constant or at least exhibiting slow drifts: It results to very low prediction accuracy when it is applied in any of our domains.

The proposed RMF* method includes significant modifications and enhancements of the base RMF algorithm producing in real time the next k forward positions, using minimal storage and processing resources. It exploits dynamic motion pattern matching interchangeably with linear-only modes of operation. RMF* incorporates the advantages of linear extrapolation for the steady parts of the flights, while at the same time exploits additional information regarding any shift in the motion type provided by critical points produced by the synopses generation task, before activating the full pattern-matching mode. This means that the algorithm continuously checks for drifts to non-linear phases, i.e., the beginning of turn and/or altitude change, activating the proper differential approximation method accordingly, including sections of circular, ellipsoid, parabolic, hyperbolic or general quadratic trajectory.

RMF* can achieve very accurate predictions for the FLP task, as the data effectively capture the dynamics of the trajectory. Since FLP is very difficult for the take offs and landings, the experimental evaluation of the proposed RMF* is primarily focused on the aviation domain and specifically on these non-linear phases. Results provided in Figure 5(a) are based on complete flights between two airports (Barcelona-Madrid) and present average 2-D spatial error (longitude, latitude) of roughly 1-1.2 km for a look-ahead time frame of up to a minute, with a sampling rate of 8 secs, and 8 look ahead steps, i.e., roughly a 1-minute look-ahead time window (mean \approx 1000m, stdev \approx 500m, skewed towards zero).

The proposed RMF* algorithm is under fine-tuning of the pattern-matching module, with special attention in identifying and modelling a set of motion patterns, or “primitives”, separately for the aviation and for the maritime domains, so that the module can promptly and correctly identify the best choice when in non-linear mode.

Hybrid Clustering/HMM method for TP purposes: In order to address the TP task, there is a trend of using stochastic models for discovering and retrieving patterns from past history. Additionally, there is a need to address the task in the scale of big data. To address this challenge towards long term trajectory predictions our proposal is to partition historic trajectories into subsets, train separate predictive models for each one of them and then use these models for individual predictions, provided that the ability to select the correct model exists.

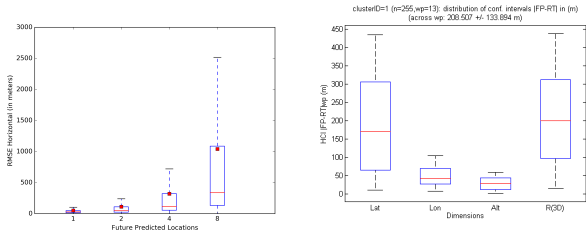


Figure 5: (Left) RMSE* prediction accuracy over various look-ahead time frames. (Right) Accuracy estimations for the per-waypoint deviation (m) from flight plan (cluster size=255) with the hybrid clustering/HMM method.

Clustering is the most popular approach for unsupervised learning, ranging from simple k nearest neighbour (k -NN) grouping, to multi-level hierarchical restructuring of the input data and using an arbitrary well-defined distance function as similarity metric. The advantages of a clustering approach include computing “cohesive” clusters of trajectories that are of smaller scale than the original set, while a distance function that exploits any data linked to enriched trajectories can be used. The SemT-OPTICS algorithm [26] is such an approach, where the similarity between two trajectory points is decomposed in two parts: The one regarding their spatio-temporal similarity and another for the enriching data, adopting an appropriate variant of Edit distance with Real Penalty (ERP) [9].

The Hidden Markov Model (HMM) approach is widely used in modelling and predicting time series, including spatio-temporal mobility patterns. The HMM approach models the evolution of an entity’s motion by a set of states and transitions between them, each one accompanied by a probability that is typically extracted by analyzing historic data. Additionally, the deviations between “intended trajectories” (e.g. flight plans in the ATM domain) and actual routes are modelled as HMM observations or *emissions*, in order to construct a probabilistic model for trajectories. We designed HMMs in a way that exploits reference points in conjunction to the enriching information. This is in contrast to approaches exploiting raw trajectory data [7, 8].

Based on these, we devised a novel Hybrid Clustering/HMM approach [12] to address the TP task, following a two-stage rationale: Clustering at the first stage of processing and training HMMs for each cluster, using only the reference points of the *medoid* of each cluster. The rationale behind this modelling approach is to be able to predict deviations from flight plans optimally, based on all the information available, including local weather (per waypoint), aircraft size, seasonal factors (time, weekday), etc. The current experiments on real aviation data (Spain, April 2016) show that deviations from flight plans can be predicted with a combined 3-D spatial accuracy of 183–736m (RMSE), in terms of half-width confidence interval for mean deviation between intended (flight plan) and actual route flown, averaged over the entire sequence of reference points for all clusters and statistically significant at $\alpha=0.05$. Results are shown in Figure 5 for one such cluster in the Madrid/Barcelona flights. Additionally, this approach exhibits at least an order of magnitude better accuracy in terms of absolute cross-track error compared to the current state-of-the-art “blind” HMM for TP, while at the same time it exhibits two to three orders of magnitude less processing

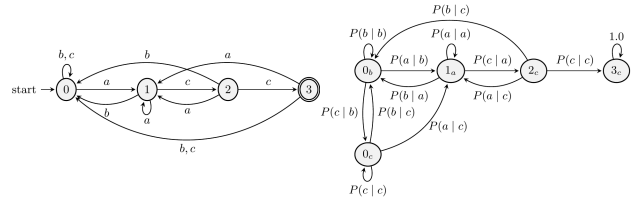


Figure 6: (Left) DFA and (Right) the corresponding Markov Chain.

and storage resources, due to the combined scaling-down of data due to clustering and to the use of reference points.

The proposed Hybrid Clustering/HMM approach is still under optimization. For the clustering stage, the challenge is to customize the similarity metrics properly and separately for the aviation and the maritime domains. For the HMM stage, the main challenge is to capture the statistics of the per-waypoint deviations for entire clusters of trajectories. Especially for the maritime domain, the reference points must be defined more dynamically (e.g. via detected critical points) since there are no equivalents to flight plans available. Hence, more specialized probabilistic distributions are tested for modelling the combination of distance-related Gaussian error distributions per-dimension. This typically involves exhaustive cross-validation experiments for prediction accuracy, rather than estimation of confidence intervals; e.g. via t-Student significance tests. Finally, segmented-trajectory models are also investigated, for very large training data sets.

6 COMPLEX EVENTS DETECTION AND FORECASTING

Given a stream of low-level events and a set of patterns defining spatial and temporal relations between low-level events, operational constraints and contextual information, we need to detect, in a timely manner, when these relations are satisfied. Whenever these relations are satisfied, we say that a high-level (or complex) event has been detected. Besides event detection, which contributes to increasing situation awareness, given the importance of predictability in both domains, we additionally address the problem of forecasting the occurrence of complex events.

Specifically, given the input stream of low-level events generated by the components described in Section 4.2, let us consider the case where a maritime analyst is interested in isolating parts of a vessel’s trajectory during which a vessel changes its direction by 180 degrees: These could indicate fishing activity. We could formally define this complex event as a temporal sequence of Change In Heading events where the first and last events in the sequence have opposite headings (headings difference is close to 180 degrees). This HeadingReversal pattern could be given to the event detection and forecasting module.

Whereas there exist multiple event detection systems, at different levels of maturity, very few of them address the issue of forecasting ([22] is one of the few cases). Moreover, being able to predict complex events which are defined by patterns that are not simple sequences of input events, poses significant challenges. Our event detection and forecasting module advances the state-of-the-art by moving beyond sequential patterns. It has the ability to predict complex events that are defined in the form of regular expressions, where the low-level events may be related through *sequence*, *disjunction* or *iteration*. In addition, by employing a

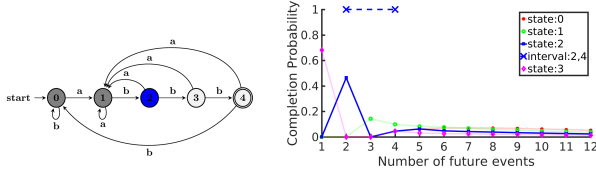


Figure 7: (Left) DFA and (Right) waiting-time distributions.

rigorous probabilistic framework, it can handle input streams that are generated by higher-order Markov processes (see [2] for a detailed description).

As a first step, event patterns in the form of regular expressions are converted to Deterministic Finite Automata (DFA). A detection occurs every time the DFA reaches one of its final states. As an example, see Figure 6 which depicts the DFA constructed for the simple sequential expression $R=acc$ (one event of type a followed by two events of type c) where the set of events that may be encountered are $\Sigma=\{a, b, c\}$.

For the task of forecasting, we need to build a probabilistic model for (the behaviour of) the DFA. We achieve this by converting the DFA to a Markov chain. If we assume that the input events are independent and identically distributed (i.i.d.), then it can be shown that we can directly map the states of the DFA to states of a Markov chain and the transitions of the DFA to transitions of the Markov chain. The probability of each transition would then be equal to the occurrence probability of the event that triggers the corresponding transition of the DFA. However, if we relax the assumption of i.i.d. events, then a more complex transformation is required, in which case the transition probabilities equal the conditional probabilities of the events. An example, see Figure 6 which shows the Markov chain derived from the indicated DFA, if we assume that the input events are generated by a 1st-order Markov process (see [2] for details). We call such a derived Markov chain a *Pattern Markov Chain* (PMC).

Once we have obtained the PMC corresponding to an initial pattern, we can compute certain distributions that are useful for forecasting. At each time point the DFA and the PMC will be in a certain state and the question we need to answer is the following: how probable is it that the DFA will reach its final state (and therefore a complex event will be detected) in k time points from now? The answer to this question depends on the state of the PMC. Hence, for each such state we need to calculate a separate distribution. These distributions are called waiting-time distributions. As an example, Figure 7 shows a DFA and the *waiting-time distributions* for its states.

In order to estimate the final forecasts, another last step is required. Forecasts are provided in the form of time intervals, like $I = (start, end)$. When such a forecast is produced, its meaning is that the DFA is expected to reach a final state sometime in the future between start and end with probability at least some constant threshold θ (provided by the user). These intervals are produced by a single-pass algorithm that scans a waiting-time distribution and finds the smallest (in terms of length) interval that exceeds this threshold. As an example, Figure 7 shows a DFA being in state 2, the waiting-time distribution for this state is shown in blue color, together with the forecast interval extracted ($I = (2, 4)$).

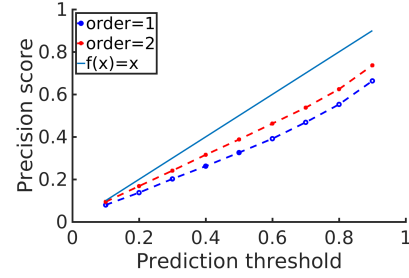


Figure 8: Precision achieved for events' forecasting using different Markov process' orders.

The above described method has been implemented in a system called Wayeb, tested with real-world maritime annotated and enriched trajectories. We show results from one pattern applied to a single vessel. The pattern is as follows:

```
R=ChangeInHeadingNorth
(ChangeInHeadingNorth+ChangeInHeading East)*
ChangeInHeadingSouth
```

where $+$ stands for disjunction and $*$ for iteration and each *turn* event has additionally been annotated with the vessel's heading. This pattern attempts to detect a NorthToSouthReversal event where a vessel executes a series of turns, initially heading towards the northern direction and eventually ends heading towards the southern direction. Figure 8 shows the precision of the proposed forecasting method for this pattern using different prediction thresholds. The precision is defined as the percentage of forecasts which were accurate (i.e. the event was indeed detected within the forecast interval). It shows results both for the assumption of a 1st-order and for a 2nd-order Markov process. We can see how increasing the assumed order does indeed positively affect precision.

Promising such results as they may be, there still remain significant challenges ahead. The most fundamental concerns *reliability*, i.e., the ability to naturally (without a pre-processing step) handle events with attributes and relations between the attributes of different events in a pattern. For example, in the NorthToSouthReversal pattern, the information about the vessel's heading would simply be an attribute which could be checked with predicates like $IsHeading(North)$. Moreover, the method that we have proposed assumes *stationarity* which implies that the transition matrix of the PMC does not change. However, the statistical properties of a stream may indeed change over time in which case we would need an efficient method for updating online the probabilistic model.

7 VISUAL ANALYTICS

The purpose of the Visual Analysis approach is to combine algorithmic analysis with the human analyst's insight and tacit knowledge in the face of incomplete or informal problem specifications and noisy, incomplete, or conflicting data [33]. Visual Analysis therefore is an iterative process where intermediate results are visually evaluated to ascertain and inform subsequent analysis steps based on prior knowledge and gathered insights. From the perspective of Visual Analytics, analysis methods fall into two categories within the overall architecture (Figure 1).

On the batch layer, Visual Analytics (VA) augments a wide range of tasks from initial data exploration and curation, complex analysis workflows, to refining and evaluating the different

models. Synoptic analysis tasks that are the subject of such exploratory visual analyses presume availability of global measures like spatial extents, value ranges, (as yet undiscovered) patterns defined over large time spans/time cycles, and thus must be supported over sufficiently large data sets. Specifically, it is worth noting that due to the exploratory focus, VA does not prescribe a rigid pipeline of algorithmic processing steps, nor does it prescribe a fixed composition of specific visualizations, as opposed to typical dashboards [3].

To cope with these requirements in an efficient and scalable way, the VA component within the integrated architecture is itself of a modular, extensible design, as shown in Figure 9. It comprises four principal component groups – data storage, analysis methods, data filtering and selection tools, and of course, visualization techniques. Different components are typically composed in an ad-hoc fashion, through visual-interactive controls, to facilitate the workflow required by the human analyst’s task at hand.

The following paragraphs review several novel workflow compositions addressing different analytical challenges in both the ATM and maritime domains.

The ability to understand data properties and to assess their quality is a crucial first step in any data analysis setting. Dealing with massive movement data analysed in context (e.g., as weather data) amplifies both the importance of that first step as well as the technical challenge involved in dealing with such large data.

Investigation of quality of movement data, due to their spatio-temporal nature, requires consideration from multiple perspectives at different scales. In paper [4], we review the key properties of movement data and, on their basis, create a typology of possible data quality problems and suggest approaches to identifying these types of problems. In particular, we systematically consider different approaches to position recording and related properties of movement data, taking into account properties of the mover set, spatial properties, temporal properties and data collection properties.

However, while [4] lays the foundation for a structured approach to detect and rectify data quality issues, cleaning and repairing data for curation purposes are still largely manual tasks that rely on a combination of tools and technologies such as database SQL, scripts, and functionality available in the VA toolkit. Especially when handling large data sets (many moving entities, long time periods) these tasks can become tedious and time-consuming. Therefore, as one facet of datAcron objective is to create advanced and scalable spatio-temporal data integration and management solutions, a modular framework is being developed that combines Big Data processing technologies with interactive visual reporting to automatically evaluate the quality of large movement data sets.

To support preparatory data analysis for building appropriate detection and prediction models, specifically patterns targeting at trajectories, events, spatial time series and spatial situations, novel methods are required that combine interactive visualizations with appropriate computational methods such as clustering, event detection, summarization and abstraction, as well as providing possibilities for manipulating parameters of computational methods and evaluating sensitivity to parameters.

In [6] we introduced the concept of time mask, which is a type of temporal filter suitable for selection of multiple disjoint time intervals in which some query conditions on arbitrary attributes hold. Such a filter can be applied to time-referenced objects, such as events and trajectories, for selecting those objects or segments

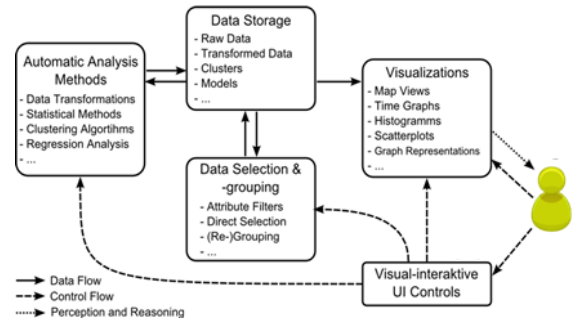


Figure 9: Principal components of the VA toolset.

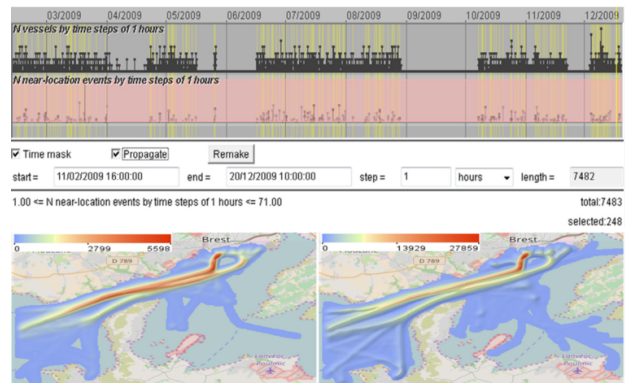


Figure 10: (Top) A time series display shows the counts of the vessels (upper row) and the near-location events (lower row) by 1-hour time steps. A query selects the intervals containing at least one event (yellow markers). (Bottom) The density of the trajectories in the times of occurrence of near-location events (left) and in the remaining times (right) [6].

of trajectories that fit in one of the selected time intervals. The selected subsets of objects or segments are dynamically summarized in various ways, and the summaries are represented visually on maps and/or other displays to enable exploration. The time mask filtering can be especially helpful in analysis of disparate data (e.g., event records, positions of moving entities, and time series of measurements), which in the considered scenarios even come from different sources.

To detect relationships between such data, the analyst may set query conditions based on one data set and investigate the subsets of objects and values in the other data sets that co-occurred in time with these conditions (e.g., see Figure 10).

Clustering of trajectories of moving entities by similarity is an important technique in movement analysis. Existing distance functions assess the similarity between trajectories based on properties of the trajectory points or segments [3]. The properties may include the spatial positions, times, and thematic attributes. There may be a need to focus the analysis on certain parts of trajectories, i.e., points and segments that have particular properties. According to the analysis focus, the analyst may need to cluster trajectories by similarity of their relevant parts only. For example, when analysing routing decisions taken by airlines in the ATM context, only the cruise phase of a flight is relevant for comparison, but not holding patterns nor takeoff and landing runway directions [5]. Throughout the analysis process, the focus

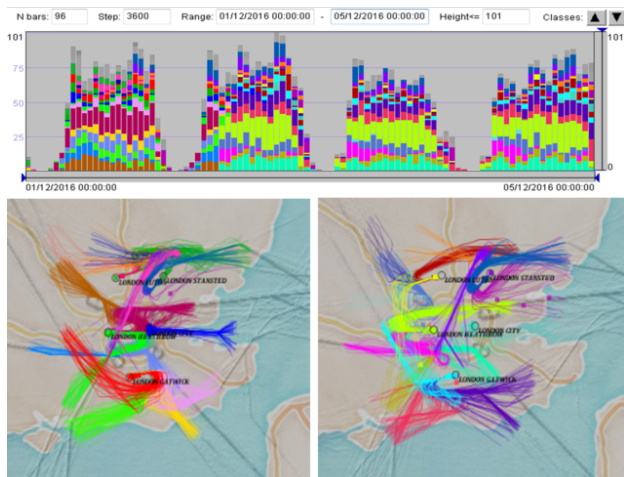


Figure 11: (Top) Bars in a time histogram show the counts of the flight arrivals in hourly intervals. Bar segments are painted in the colors of the clusters the flights belong to. (Bottom) The final parts of the flight trajectories in days 1 and 3 are colored according to the cluster membership [5].

may change, and different parts of trajectories may become relevant, e.g., due to weather conditions. In paper [5], we propose an analytical workflow that uses interactive filtering tools to attach relevance flags to elements of trajectories; subsequent clustering uses a distance function that ignores irrelevant elements. The resulting clusters are summarized for further analysis. The paper demonstrates how this workflow can be useful for different analysis tasks in three case studies related to ATM flow management (Figure 11). The paper [5] further proposes a suite of generic techniques and visualization guidelines to support movement data analysis by means of relevance-aware trajectory clustering.

For developing and evaluating trajectory prediction algorithms it is important to have the possibility of detailed comparison of predicted trajectories to actual ones, to see how accurate the prediction is. It is also necessary to compare predictions obtained with different parameter settings, to understand the impact of the parameters and to choose the most suitable settings.

On the real-time layer, in-stream processing algorithms operate directly on data streams under predefined parameter settings for monitoring purposes, i.e., trajectory & location prediction (Section 5) as well as event forecasting (Section 6). The main goal here is to provide a visual interface to the detection and prediction model output, presented in the context of real-time spatio-temporal data comprising the current situational picture (vessel trajectories, specific areas, weather information etc.). These visualizations provide a limited set of interaction for confirmatory analysis of detected outliers and patterns, as well as in-context validation of model predictions, and typically are offered as dashboard components.

A novel technique is the point matching method that is supplemented by interactive visual interfaces enabling the analyst to view and explore the results of point matching (Figure 12).

For the purposes of situation monitoring, a real-time visualization approach has been developed as an endpoint in the

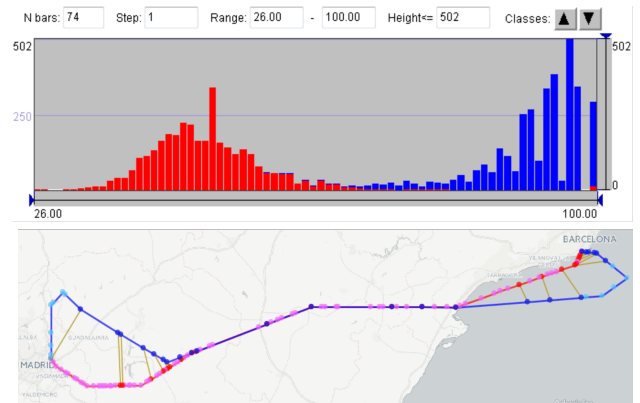


Figure 12: Detail view of a significantly mismatched pair of actual (blue) vs. predicted (red) trajectories. The histogram shows the statistical distribution of the proportions of the matched points; the map shows the spatial footprints of both trajectories.

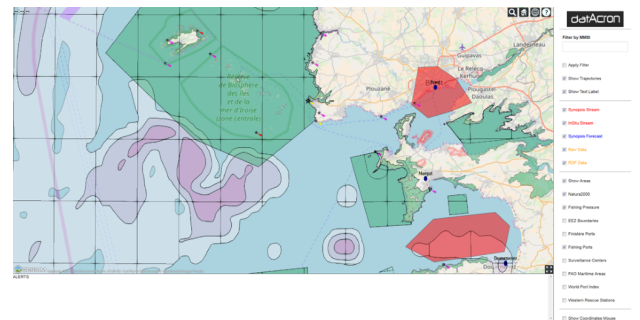


Figure 13: Real-time visualization dashboard for maritime situation monitoring.

Kafka-based communication infrastructure (Figure 13). The visualization visually encodes a selectable subset of information layers from the enriched stream provided by the data manager. This stream, as described in previous sections, includes pre-processed position data (i.e., trajectory synopses), dynamic and static context information (e.g., weather conditions, maritime areas), trajectory and location predictions, as well as detected and forecasted events (e.g., initiation of a turn manoeuvre, danger of collision).

Further work will evaluate which visual encodings and interaction capabilities (e.g., to interactively adjust event detection parameters) best match different use case requirements in both domain.

8 CONCLUSIONS

Given the user-defined challenges and the fact that more and more data of different nature and purposes is generated in the air traffic management and the maritime domains, this article reports on significant progress achieved in datAcron towards the real-time processing and analysis of big data for improving the predictability of trajectories and events regarding moving entities in those domains.

Albeit the progress, there are also significant challenges ahead in both domains: Discovery of spatio-temporal relations among entities in a timely manner, efficient query answering of very large knowledge graphs for online and offline analytics tasks,

cross-streaming synopses generation, long-term online full trajectory predictions and improvements in forecasting complex events together with learning/refining their patterns by exploiting examples; as well as, the provision of online visual analytics workflows.

ACKNOWLEDGMENTS

This work is supported by project datAcron, which has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 687591.

REFERENCES

- [1] Ibrahim Abdelaziz, Razen Harbi, Zuhair Khayyat, and Panos Kalnis. 2017. A Survey and Experimental Comparison of Distributed SPARQL Engines for Very Large RDF Data. *Proc. VLDB* 10, 13 (2017), 2049–2060.
- [2] Elias Alevizos, Alexander Artikis, and George Paliouras. 2017. Event Forecasting with Pattern Markov Chains. In *Proc. 11th ACM Intl. Conf. on Distr. and Event-based Systems*. ACM, 146–157.
- [3] Gennady Andrienko, Natalia Andrienko, Peter Bak, Daniel Keim, and Stefan Wrobel. 2013. *Visual Analytics of Movement*. Springer.
- [4] Gennady Andrienko, Natalia Andrienko, and Georg Fuchs. 2016. Understanding movement data quality. *Journal of Location Based Services* 10, 1 (2016), 31–46.
- [5] G. Andrienko, N. Andrienko, G. Fuchs, and J.M.C. Garcia. 2018. Clustering Trajectories by Relevant Parts for Air Traffic Analysis. In *IEEE Transactions on Visualization and Computer Graphics (proceedings IEEE VAST 2017)*, Vol. 24(1).
- [6] Natalia Andrienko, Gennady Andrienko, Elena Camossi, Christophe Claramunt, Jose Manuel Cordero Garcia, Georg Fuchs, Melita Hadzagic, Anne-Laure Joussemme, Cyril Ray, David Scarlatti, and George Vouros. 2017. Visual exploration of movement and event data with interactive time masks. *Visual Informatics* 1, 1 (2017), 25–39.
- [7] Samet Ayhan and Hanan Samet. 2016. Aircraft Trajectory Prediction Made Easy with Predictive Analytics. In *Proc. 22Nd ACM SIGKDD*. New York, NY, USA, 21–30.
- [8] Samet Ayhan and Hanan Samet. 2016. Time Series Clustering of Weather Observations in Predicting Climb Phase of Aircraft Trajectories. In *Proc. 9th ACM SIGSPATIAL IWCTS*. 25–30.
- [9] Lei Chen and Raymond Ng. 2004. On the Marriage of Lp-norms and Edit Distance. In *Proc. 30th VLDB*. 792–803.
- [10] Christophe Claramunt, Cyril Ray, Elena Camossi, Anne-Laure Joussemme, Melita Hadzagic, Gennady L. Andrienko, Natalia V. Andrienko, Yannis Theodoridis, George A. Vouros, and Loïc Salmon. 2017. Maritime data integration and analysis: recent progress and research challenges. In *Proc. 20th Inter. Conf. on Extending Database Technology, EDBT*. 192–197.
- [11] Anastasia Dimou, Miel Vander Sande, Pieter Colpaert, Ruben Verborgh, Erik Mannens, and Rik Van de Walle. 2014. RML: A Generic Language for Integrated RDF Mappings of Heterogeneous Data. In *Proc. 7th Work. Linked Data Web*.
- [12] X. Georgiou et al. submitted. Semantic-aware Aircraft Trajectory Prediction. (submitted).
- [13] A-L. Joussemme, C. Ray, E. Camossi, M. Hadzagic, C. Claramunt, K. Bryan, E. Reardon, and M. Ilteris. April 2016. *Maritime Use Case description, H2020 datAcron project deliverable D5.1*. Technical Report.
- [14] Kostis Kyzirakos, Ioannis Vlachopoulos, Dimitrios Savva, Stefan Manegold, and Manolis Koubarakis. 2014. GeoTriples: A Tool for Publishing Geospatial Data As RDF Graphs Using R2RML Mappings. In *Proc. ISWC 2014, Posters & Demonstrations Track - Volume 1272*. 393–396.
- [15] Maxime Lefrançois, Antoine Zimmermann, and Noorani Bakerally. 2017. A SPARQL Extension for Generating RDF from Heterogeneous Formats. In *The Semantic Web*. Springer, 35–50.
- [16] Xuelian Lin, Shuai Ma, Han Zhang, Tianyu Wo, and Jinpeng Huai. 2017. One-pass Error Bounded Trajectory Simplification. *PVLDB* 10, 7 (2017), 841–852.
- [17] Jiajun Liu, Kun Zhao, Philipp Sommer, Shuo Shang, Brano Kusy, and Raja Jurdak. 2015. Bounded Quadrant System: Error-bounded trajectory compression on the go. In *Proc. 31st ICDE*. 987–998.
- [18] Cheng Long, Raymond Chi-Wing Wong, and H. V. Jagadish. 2014. Trajectory Simplification: On Minimizing the Direction-based Error. *PVLDB* 8, 1 (2014), 49–60.
- [19] Nathan Marz and James Warren. April, 2015. *Big Data: Principles and best practices of scalable realtime data systems*. Manning Publications Co.
- [20] Nirvana Meratnia and Rolf A. de By. 2004. Spatiotemporal Compression Techniques for Moving Point Objects. In *Proc. EDBT 2004*, Elisa Bertino, Stavros Christodoulakis, Dimitris Plexousakis, Vassilis Christophides, Manolis Koubarakis, Klemens Böhm, and Elena Ferrari (Eds.). 765–782.
- [21] Jonathan Muckell, Paul W. Olsen, Jeong-Hyon Hwang, Catherine T. Lawson, and S. S. Ravi. 2014. Compression of trajectory data: a comprehensive evaluation and new approach. *Geoinformatica* 18, 3 (2014), 435–460.
- [22] Vinod Muthusamy, Haifeng Liu, and Hans-Arno Jacobsen. 2010. Predictive Publish/Subscribe Matching. In *Proc. 4th ACM International Conference on Distributed Event-Based Systems*. ACM, 14–25.
- [23] Axel-Cyrille Ngonga Ngomo. 2013. ORCHID – Reduction-Ratio-Optimal Computation of Geo-spatial Distances for Link Discovery. In *Proc. of ISWC 2013*. 395–410.
- [24] P. Nikitopoulos, C. Doukeridis, A. Vlachou, and G.A. Vouros. 2018. DiStRDF: Distributed Spatio-temporal RDF Queries on Spark. In *BMDA workshop*.
- [25] Kostas Patroumpas, Elias Alevizos, Alexander Artikis, Marios Voudas, Nikos Pelekis, and Yannis Theodoridis. 2017. Online event recognition from moving vessel trajectories. *Geoinformatica* 21, 2 (2017), 389–427.
- [26] Nikos Pelekis, Stylianos Sideridis, Panagiotis Tampakis, and Yannis Theodoridis. 2016. Simulating Our LifeSteps by Example. *ACM Trans. Spatial Algorithms Syst.* 2, 3 (2016), 11:1–11:39.
- [27] Michalis Potamias, Kostas Patroumpas, and Timos Sellis. 2006. Sampling Trajectory Streams with Spatiotemporal Criteria. In *Proc. of 18th SSDBM*. IEEE Computer Society, 275–284.
- [28] Georgios M. Santipantakis, George A. Vouros, Christos Doukeridis, Akrivi Vlachou, Gennady Andrienko, Natalia Andrienko, Georg Fuchs, Jose Manuel Cordero Garcia, and Miguel Garcia Martinez. 2017. Specification of Semantic Trajectories Supporting Data Transformations for Analytics: The datAcron Ontology. In *Proc. of 13th International Conference on Semantic Systems (Semantics2017)*. ACM, 17–24.
- [29] Mohamed Ahmed Sherif, Kevin Dreßler, Panayiotis Smeros, and Axel-Cyrille Ngonga Ngomo. 2017. Radon - Rapid Discovery of Topological Relations. In *Proc. 31st AAAI Conference on Artificial Intelligence*. 175–181.
- [30] Panayiotis Smeros and Manolis Koubarakis. 2016. Discovering Spatial and Temporal Links among RDF Data. In *Proc. of LDOW@ WWW*.
- [31] Stefano Spaccapietra, Christine Parent, Maria Luisa Damiani, Jose Antonio de Macedo, Fabio Porto, and Christelle Vangenot. 2008. A Conceptual View on Trajectories. *Data Knowl. Eng.* 65, 1 (2008), 126–146.
- [32] Yufei Tao, Christos Faloutsos, Dimitris Papadias, and Bin Liu. 2004. Prediction and Indexing of Moving Objects with Unknown Motion Patterns. In *Proc. SIGMOD International Conference on Management of Data*. ACM, 611–622.
- [33] James J. Thomas and Kristin A. Cook. 2005. *Illuminating the Path: The Research and Development Agenda for Visual Analytics*. IEEE Computer Society, Los Alamitos, CA, United States(US).
- [34] Yang Yang, Jun Zhang, and Kai-quan Cai. 2015. Terminal-Area Aircraft Intent Inference Approach Based on Online Trajectory Clustering. (07 2015), 1–11.

Scouter: A Stream Processing Web Analyzer to Contextualize Singularities

Badre Belabess
Atos Innovation Lab
Bezons, France
badre.belabess@atos.net

Musab Bairat
Atos Innovation Lab
Bezons, France
musab.bairat@atos.net

Jeremy Lhez
LIGM, CNRS, ENPC, ESIEE, UPEM
Marne-la-Vallée, France
jeremy.lhez@u-pem.fr

Zakaria Khattabi
Atos Innovation Lab
Bezons, France
zakaria.khattabi@atos.net

Yufan Zheng
Atos Innovation Lab
Bezons, France
yufan.zheng@atos.net

Olivier Curé
LIGM, CNRS, ENPC, ESIEE, UPEM
Marne-la-Vallée, France
olivier.cure@u-pem.fr

ABSTRACT

Anomaly detection is a key feature of applications processing singularities using IoT sensor measures. In a recent project, we highlighted that to guarantee high quality detections, metadata providing spatio-temporal contexts on sensor measures are needed. These metadata hence become an integral component of the anomaly detection computation and need to be processed using a streaming approach. In this paper, we introduce Scouter, a generic tool that helps in capturing, analyzing, scoring and storing the contextual information of a given application domain. The processing depends on a semantic-based approach that exploits ontologies to score the relevancy of contextual information. The paper provides details on the system's architecture, describes lessons learned and introduces practical aspects.

1 INTRODUCTION

Large amounts of data generated by IoT streaming devices are continuously accumulated and processed in Big Data platforms. The analysis of these data supports intelligent software functionalities usually based on machine learning or semantic approaches. Among these features, singularities leading to anomaly detection is predominant and is tackling domains as diverse as medicine (e.g., to identify malignant tumors in MRI images), finance (e.g., to discover cases of credit card transaction frauds), information technology (e.g., to detect hacking situations in computer networks). In the WAVES project¹, we are interested in detecting anomalies in large potable water networks managed by an international company, i.e., Suez company, referred to as the domain field expert in the paper. The automatic detection of such anomalies is an important issue at both the environmental and economic levels: the volume of lost water in the world has reached more than 32 billions m³/year (corresponding to a loss of US\$ 14 billion/year) with 90% of them being invisible due to the underground nature of the network. As a matter of fact, water leaks can be detected based on pressure and flow measures retrieved from sensors installed on strategic spots within such a network. During several experiments, conducted on the French network, consisting of 100.000 km of pipelines equipped with over 3.000 sensors distributing drinkable water to more than 12 million clients, field

experts found out that to guarantee high accuracy in anomaly detection, a contextualization of the measures is needed especially when a singularity appears. For instance, abnormal high pressure or peculiar flow signatures could potentially indicate a water leak. However, in many cases of popular sport or cultural events, these singularities could easily be explained because of some arrangements made by the city hall to provide water fountains. The same singularities can also account for hot weather conditions implying garden watering in a suburb area or a wildfire forcing firemen to use important amounts of water. Hence, an efficient approach for singularity contextualization has to integrate a spatio-temporal dimension on the analyzed measures.

The tasks related to the capturing, analyzing, scoring and storing of contextual metadata are demanding since they require interactions between software components that may be hard to set up, especially in a distributed environment. These components generally handle stream, natural language and ontology processing as well as the storing of complex data structures. Designed as a generic system, Scouter² aims at simplifying these tasks by proposing implementations of the main functionalities and by taking care of installation and configuration aspects.

2 CONTRIBUTIONS

In this paper, we present Scouter, a novel singularity contextualization system built on top of Apache Kafka that automatically proposes relevant explanations to detected anomalies using a continuous filtering approach.

Scouter proposes a powerful web analysis tool leveraging on ontology building and semantic web technologies to automatically retrieve, score and rank relevant events extracted from the web. Scouter also exposes a powerful set of natural language processing functions such as topic extraction, topic relevancy, topic matching and sentiment analysis as well as an original method for geo-profiling to classify areas. Finally, Scouter reduces the time complexity of heavy web scrapping using big data frameworks and eliminates the dataset size limitation by implementing a continuous filtering method. In summary, the main contributions of Scouter are:

- An efficient method to fetch both streaming and static data from various sources on the web based on a hierarchy graph of concept labels, henceforth denoted an ontology.
- An original geo-profiling method that gives a detailed portrayal of the areas where the events occur.

¹<https://www.waves-rsp.org/>

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

²<https://badrebelabess.github.io/Scouter-1/>

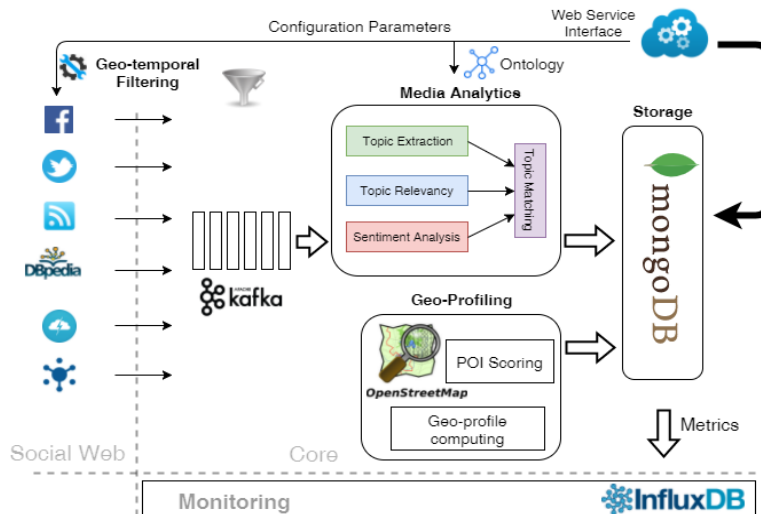


Figure 1: Scouter Architecture

- A novel approach to select relevant non duplicate events using powerful Natural Language Processing (NLP) functions in a three dimensional pipeline: topic extraction, topic relevancy and sentiment analysis.
- A continuous dynamic data filtering technique based on a combination of media analytics scoring and geo-profiling annotations.

3 ARCHITECTURE

As a component of the WAVES platform [1], Scouter was developed to be a generic system that can handle both streaming and static events and analyze them using a powerful set of NLP functions and semantic methods. Fully configurable, the primary goal of Scouter is to fetch data efficiently from different web sources, process them in a short amount of time in order to score every event regarding its capability to explain anomalies detected by the platform. As detailed by Figure 1, the main components of our system are: a set of Web data connectors, a media analytics unit, a geo-profiling unit, a storage mainframe, a messaging broker and a web service provider.

The web connectors consume data from different data sources at a certain frequency based on predefined configurations which can be specified using a web interface. These sources include social networks namely Twitter and Facebook (*e.g.*, citizens commenting on water leaks nearby) but also media sources such as RSS feeds from various newspapers listed in Table 1 (*e.g.*, an article from the French newspaper Le Monde mentioning a fire), weather information from Open Weather Map (*e.g.*, climate conditions during a specific event), organized events from Open Agenda (*e.g.*, concerts, exhibitions or sporting events) and also specific information retrieved from DBpedia (*e.g.*, number of inhabitants or type of neighborhoods). All of these data sources are consumed in a powerful multi-threading mechanism using rest APIs. The concepts, subconcepts and properties used to fetch data are represented in an ontology that formalizes the different relationships in a sound manner. More details about the ontology are provided in the section 4.1.

The media analytics unit digests fetched feeds from Kafka and leverages on the Apache Spark distributed framework to analyze feeds in real-time. Feeds are recorded as events annotated with

location, start/end dates and description. In order to filter the most relevant events without any duplicates in the database, a topic extraction approach and a sentiment analysis are combined to match topics. The topic extraction parses the text of feeds to discover term occurrences. Then, the scoring module takes advantage of user defined weights, *i.e.*, a real value in the $[0, 1]$ range, associated to ontology concepts to provide an overall scoring for each text. The sentiment analysis classifies the feeds into positive or negative categories using the maximum entropy algorithm [2]. It builds a model using multinomial logistic regression to determine the right category for a given text. Simultaneously, the geo-profiling unit provides geographical characteristics for the analyzed area. It determines the type of terrain surrounding the anomaly location. Based on points of interest and terrains for a given zone, a profile is generated that describes the category of the targeted region using a configurable classification.

After the scoring step, events are recorded into a scalable and distributed document database (namely MongoDB). As a result, we obtain in real-time spatio-temporal and scored contexts that can assist the operator to explain an anomaly detected in the underground water pipeline network. Scouter also provides a metrics monitoring tool to track the performance of the system including query times, event processing times, events count and topic extraction training times. These metrics are stored in a time series database with very high read/write access (namely InfluxDB). Finally, the Web services component is used for configuring the system. It provides Rest-based interface that can be integrated with a graphical user interface to deliver configuration parameters in an user-friendly and readable way.

4 MEDIA ANALYTICS & NLP

In this section, we detail the methodology of collecting data from the various types of sources available on the web. This process is built on four major components that work together in order to:

- Extract the most relevant events based on a complex graph of concepts and properties.
- Identify the appropriate summaries from each event with the maximum expressiveness.

- Avoid duplicate events stored in the database by comparing summaries and assessing the similarity between them using sentiment analysis.

4.1 Ontology

Every scrapping tool relies on a configuration file that lists the properties of words, concepts or events that it tries to fetch [3]. In Scouter’s case, the fetching capabilities rely heavily on a pre-built ontology that lists the main concepts the user is looking for but also organizes the relations in two dimensions:

Vertical Hierarchy: A given concept (e.g., Fire) can have multiple sub-concepts (e.g., Blaze, Wildfire) or aliases and misspellings (e.g., fir, wild-fire, blayz).

Horizontal Dependency: A concept can have multiple properties that describe a specific state in a certain time period. For example, water can be potable, but can also leak or have a specific color.

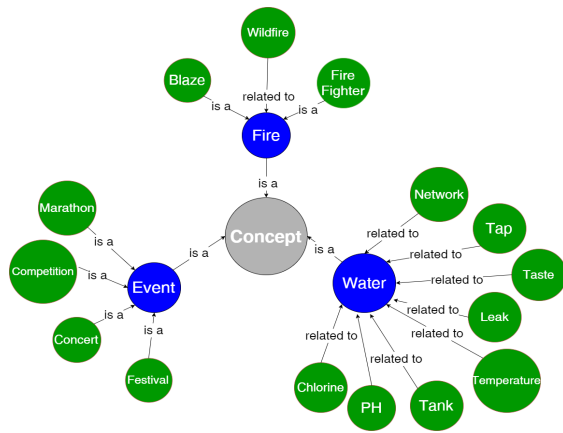


Figure 2: Ontology Overview

By combining the concepts and the properties through predicates, we can build a complex graph such as the one in Figure 2 used for the water leak use case. We can easily argue that this type of structure holds more expressiveness than a classic list of keywords exposed in a configuration file.

4.2 Topic Extraction

After fetching the proper events from the various sources based on the ontology of concepts and subconcepts, the next step is to extract meaningful topics from the events following the pipeline in in Figure 3.

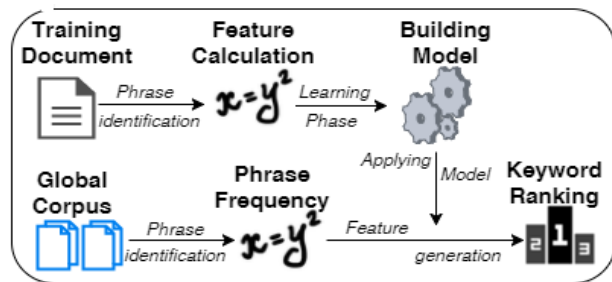


Figure 3: Topic Extraction Pipeline

The main preprocessing here concerns the cleaning of the input text, the identification of potential candidates, and finally the

stemming and case-folding of the phrases. Input files are filtered to regularize the text and determine initial phrase boundaries, then the splitting into tokens alongside several modifications are made (apostrophes are removed, hyphenated words are split in two, etc). Next, we consider all the subsequences in order to determine the ones that are suitable candidate phrases. To increase the accuracy, we use a list of french stop-word list containing more than 500 words in different syntactic classes (conjunctions, articles, particles, etc). Then we case-fold all words and stem them using the iterated Lovins method [4] to discard any suffix, and repeating the process until there is no further change. Stemming and case-folding allow us to treat different variations on a phrase as the same thing.

The main processing involves two calculated features for each candidate phrase : the phrase frequency in the input text compared to its rarity in general use and the first occurrence, which is the distance into the input text of the phrase first appearance. These two features are converted to nominal data for the machine-learning process and a discretization table for each feature is derived from the training data. Finally, we generate a model that gives the scores for every candidates and ranks them using Naive Bayes techniques [5].

4.3 Topic Relevancy

Several research works tackle the issue of automatic summarization [6]. The tools proposed generally mix several classes of features such as summary likelihood, use of topic signatures or syntactic analysis [7]. In our case, we chose a direct approach based on distributional similarity that compares input and summary content. In fact, we consider that a good summary should be characterized by low divergence between probability distributions of words in the input and summary, and by high similarity with the input. For this purpose, we used two common measures: the Kullback Leibler divergence and the Jensen Shannon divergence.

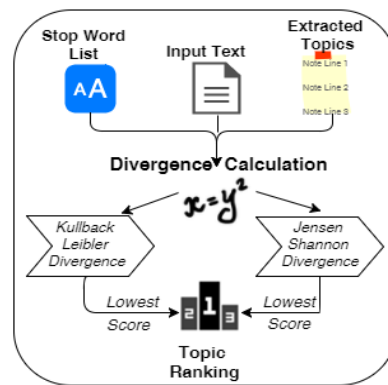


Figure 4: Topic Relevancy Pipeline

First, words in both input and summary are stemmed and separated before any computation. Then we compute the two measures:

Kullback Leibler (KL) divergence: It corresponds to the average number of bits wasted by coding samples belonging to P using another distribution Q, an approximate of P. It is given by:

$$D_{KL}(P||Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)}.$$

In our case, the two distributions of word probabilities are estimated from the input and summary, respectively. Because KL divergence is not symmetric, both input summary and summary input divergences are introduced as metrics. In addition, we perform a simple smoothing using an approximating function that captures important patterns while leaving out noise and other fine-scale structures.

Jensen Shannon (JS) divergence: This one leverages on the fact that the distance between two distributions cannot be very different from the average of distances of their mean distribution. It is given by the following formula:

$$JSD(P \parallel Q) = \frac{1}{2}D(P \parallel M) + \frac{1}{2}D(Q \parallel M) \quad (1)$$

$$\text{where } M = \frac{1}{2}(P + Q) \quad (2)$$

In contrast to KL divergence, the JS distance is symmetric and always defined. We compute both smoothed and unsmoothed versions of the divergence as summary scores. The final step is to use the output of these two functions to rank the extracted topics and keep only the ones with the best summarization score (*i.e.*, lowest divergences).

4.4 Sentiment Analysis

During the last decade, sentiment analysis has known an exponential development due to the growing usage of social networks and the popularity of websites where people can state their opinion on different products and rate them. Many solutions have been proposed and packaged in several technologies [8], we propose in this section a simple approach based on some tools provided by the Stanford CoreNLP toolkit [9].

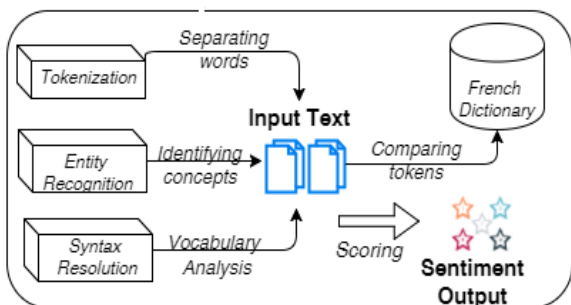


Figure 5: Sentiment Analysis Pipeline

Before applying the model, we need to launch several preprocessing steps that will improve the accuracy of the output score. Here, three steps are involved:

Tokenization: It separates the text into a sequence of tokens and saves the character offsets of each token in the input text. Then we split a sequence of tokens into meaningful sentences.

Entity Recognition: It checks if the tokens are consistent and conform to a predefined standard before trying to determine the likely gender information to names based on a dictionary. Then, the recognition algorithm annotates recognized tokens as persons, locations, organizations, numbers, dates, times or durations.

Syntactic Resolution: The system used a full syntactic analysis, including both constituent and dependency representation, based on a probabilistic parser.

Since our use case is to analyze media and social networks within the French territory, we used a French dictionary embedded in a wrapper to analyze the words.

After the preprocessing phase, we enter the main computation step where we apply the model. Among several models, we chose the compositional one over trees using deep learning. It relies on nodes of a binarized tree of each sentence, including, in particular, the root node of each sentence, that are given a sentiment score. In order to capture the sentiment of an input text, a Recursive Neural Tensor Network model (RNTN) is built based on the characteristics of the input phrases. These phrases are represented using word vectors and a parse tree, then we compute vectors for higher nodes in the tree using the same tensor-based composition function. This approach is inspired from the recursive deep models for semantic compositionality developed by Stanford[10].

4.5 Topic Matching

As stated at the beginning of this section, the goal of the different steps in the media analytics part is to extract the most relevant unique events by annotating them with a meaningful summary. Hence, we try to avoid duplicate events that refer to the same happening or occurrence. In order to complete this task with high accuracy, we are mixing several approaches relying on NLP processing from ontology building to sentiment analysis. The process follows the following steps: For each event fetched from the different sources, the topic extraction phase will propose a list of potential summaries based on a Bayesian approach. Then these summaries will be ranked using the lowest divergences (*i.e.*, KL divergence and JS divergence) in order to assess their accuracy. Among the highest ranked ones, we will check if they have the same sentiment (*i.e.*, positive, neutral or negative). If one of the selected topics during this process have the same sentiment, we assume then that they are referring to the same event in the same way. Therefore, we conclude that these events are duplicates and we only keep the content of one event. Also, we annotate the event with a reference from the other deleted event to show to the final user that this specific event is present in different sources. The overall pipeline of our media analytics module is detailed in figure 6.

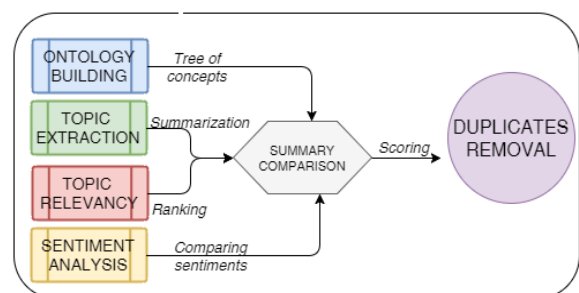


Figure 6: Topic Matching Pipeline

Even though this module provides powerful functions to filter relevant unique even, a spacial dimension is required to fully contextualize the detected anomaly. This part will be explained in the following section.

5 GEO-PROFILING

5.1 Methodology

The goal of Scouter is to provide relevant information to contextualize and potentially explain detected anomalies. The media analytics module helped in filtering the relevant events and removing duplicates, however geographical information about the area where the anomaly is spotted could further fine-tune and improve the accuracy of the context. This task is handled by two complementary modules: Geo-profiling module and reasoning module. It can be performed before the reasoning, to orientate the research of events, or after, to change the ranking of the potential sources. It is composed of two methods, that are combined and enriched with a third consumption-based method for better results.

Method 1: It extracts points of interest (POI) present in a given sector, from online geographic data sources, to determine the proportion of different types of surfaces composing it. The domain field expert selected the types of surfaces relevant to our use cases, giving us five profiling parameters: residential, natural, agricultural, industrial and touristic. Then, we created a rating file, assigning notes to each POI, in order to compute a score for each type of surface, and calculate its proportion (*i.e.*, a real value in the $[0,1]$ range).

Method 2: It is also based on geographic data, but uses features modeled as polygons instead of POI. The inclusion tests are more complete, since some polygons may be included completely or partially inside the consumption sector. Also, the computation is not performed using the rating system, but the areas of the polygons, which are less arbitrary. Otherwise, both methods produce the same result: proportions (also as real value in the $[0,1]$ range) for each type of surface.

Method 3: For certain types of consumption sectors, our two methods can slightly differ. To decide which method should be used in each case, we added a third method that computes what we denote as the consumption ratio. For each sector, we compute the daily flow, and make an average over a long period of time to avoid anomalies; then we divide this flow by the pipeline length on the sector to obtain the ratio. A low ratio corresponds to a sector with few consumers, such as countryside zones, a high ratio is the opposite. The program selects the best profiling using those criterion.

In case of a mixed result, we compute the average of the methods, to reflect the fact that there may be both open zones and populated locations. In Figure 7, we sum up our profiling strategies by highlighting their collaborations and the input data for each method.

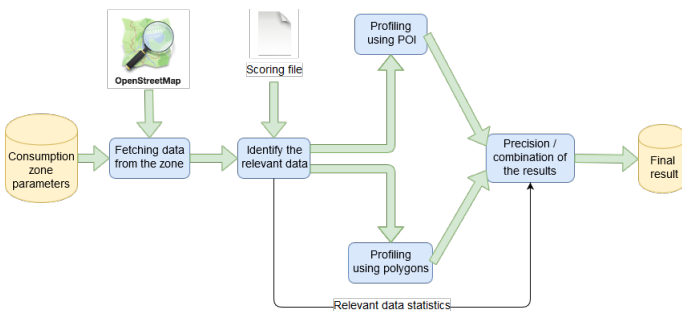


Figure 7: Profiling Overview

5.2 Tools & Resources

The geo-profiling module relies essentially on open data: the identification of the anomalies sources can only be performed with external sources, and so is their probabilistic classification.

The origins of the problems detected are generally external events that affect the quality parameters on the network. Several types of events can be responsible such as natural disasters, sport events or cultural exhibition. Some are not predictable and will be noticed by mining the proper sources, but others can be detected easily: sport events are announced a few weeks in advance, a cultural festival may be scheduled several months before. Some events can also be taken in account, even though they are not directly causing anomalies: meteorological data will help in deciding the relevancy of other events (*i.e.*, an open air festival will be more relevant as a cause of anomaly in the case of a warm weather). Static data may also provide accurate information such as Open Street Map. It has been selected because of its relative completeness compared to other online data like GeoNames. In addition to those geographic information, the Linked Open Data also provides a great amount of details on locations surrounding our water network. Some are useful to enhance the profiling or the classification of sources (*e.g.*, population density, touristic attractions or GDP), others are potential original events (*e.g.*, religious celebrations or regular typical events).

6 EVALUATION

In this section, we discuss the performance metrics used in our system, we evaluate the media analytics part on several dimensions such as the quality of events collected, then we focus on the geo-profiling module.

6.1 Media Analytics

In this section, we refer to an experimentation collecting for 9 hours events (feeds) from all the mentioned sources namely social networks (Facebook and Twitter), newspapers (RSS feeds), Open Agenda and DBpedia for organized events, Open Weather Map for climate conditions. The target geographical area of interest is a group of cities in the suburb of Paris, France, denoted as Versailles and having a coordinates bounding box. The parameters used for each of the data sources are explained in Table 1.

Source	Fetch Frequency	Pages of Interest	Concepts Scores
Facebook	12 hours	Mon Versailles Versailles Officiel Public Events	meter:1 damage:10
Twitter	Streaming	@Versailles @monversailles @prefet78 #sdis78	concert:10 spray:1 fire:10 water:10 blaze:1 wildfire:10
Open Agenda	24 hours		flow:5 tank:1 chlore:5
Open Weather Map	4 hours		
DBpedia	24 hours		
RSS News Papers	12 hours	Le Parisien 78 Actu versailles.fr Sdis78 yvelines.gouv.fr	pressure:5

Table 1: Data Sources and Concepts Scores

For instance, Twitter was run using a streaming API over all the tweets located in the bounding box, a special attention was given to extracted events from some feeds (e.g., @Versailles or @monversailles) since they are official accounts of the city. The keywords used to query these tweets are a set of 12 concepts (each one having sub-concepts in the ontology) with an affected weight that scores the relevancy of the keyword.

For this evaluation, the system is evaluated in terms of the following measures:

Number of collected and stored events: It represents the number of received feeds from all the data sources during the run time. Figure 8 shows the number of events collected during the run and the number of events stored in the database that have a score higher than 0. We can clearly see that many of the collected events (around 28% for this experiment) are not relevant, therefore they will be useless for the operator as a potential explanation for the anomalies.

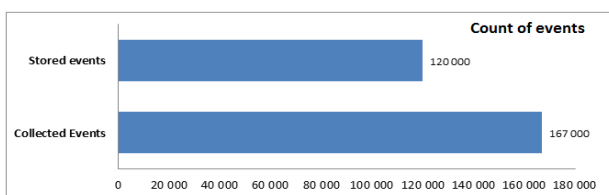


Figure 8: Collected & Stored Events for 9 Hours

Kafka Queue Messages Per Second: It represents the number of messages (events) written to Kafka broker, it can indicate the load on the messaging queue. When Scouter is running, all processors start ingesting data, then each of them will sleep until the next round after certain frequency. This explains the peak at the starting time of the figure 9, while after that, only Twitter stream feeds are being written to Kafka queue.

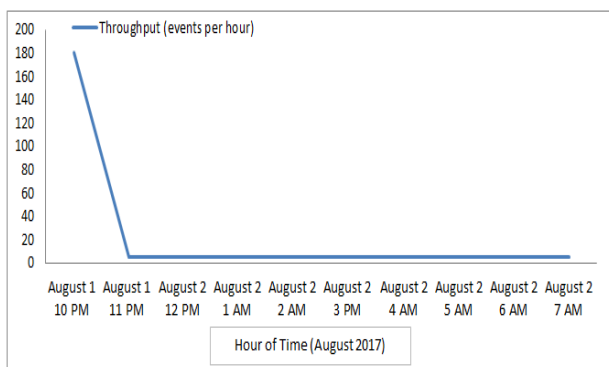


Figure 9: Kafka Throughput

Topic extraction training and Average Processing time: They represent the time spent on building the training model for detecting topics and processing the incoming events.

Table 2 shows the average time needed to score all collected events, it is calculated by dividing the sum of scoring time for each of the events by the collected events count. It also shows the training time for topic extraction algorithm to build the model that is used to detect topics. We can see that Scouter performs very well with relatively large number of events coming to the system without any failure or delay.

Measure	Time in Millisecond
Average Processing Time	7.43
Topic Extraction Training Time	474

Table 2: Scouter Processing Time

6.2 Quality of Events Collected

To evaluate the quality of the events collected, we considered the water leaks use case, where the events were fetched from the mentioned data sources over 9 hours, using the ontology in Figure 2 and the assigned score listed in Table 1. The domain expert provided the time stamp and location of all the anomalies reported on 2016 which came to 15 in total. From the database, we fetched all stored events close to the time stamp and location of each anomaly and presented them to five domain experts. For each event, they were asked if they believe that this event can give a proper and relevant explanation for the anomaly reported. A constraint was imposed, the answer had to be binary, i.e., Yes or No, in order to simplify the interpretation of the results.

Table 3: Domain Experts Evaluation

Eval- uator	Events														
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	×	✓	×	✓	✓	✓	×	✓	×	×	✓	×	×	×	×
2	×	✓	×	✓	✓	×	×	✓	×	✓	✓	×	×	×	×
3	×	✓	×	✓	✓	×	✓	×	×	✓	×	✓	✓	×	×
4	×	✓	×	✓	✓	×	×	✓	×	✓	×	×	✓	×	×
5	×	×	×	✓	×	×	×	✓	×	×	✓	×	×	×	×

To evaluate the reliability of the annotation, we used Fleiss kappa measure [11]. It is a statistical measure that aims to assess the reliability of agreement between a certain number of annotators when assigning labels to categorical subjects. It can be interpreted as expressing the extent to which the observed amount of agreement among raters exceeds what would be expected if all raters made their ratings completely randomly. It follows the equation below with the calculated results for our 5 evaluators scenario:

$$kappa = \frac{\bar{P} - \bar{P}_e}{1 - \bar{P}_e} = \frac{0.84 - 0.5256888889}{1 - 0.5256888889} = 0.6626686657$$

$$where \bar{P}_e = \sum_{j=1}^k P_j^2 = \sum_{j=1}^k \left(\frac{1}{N} \sum_{i=1}^N n_{ij} \right)^2$$

$$And P_i = \frac{1}{n(n-1)} \sum_{j=1}^k n_{ij}(n_{ij} - 1)$$

P is the mean of P_i , the extent to which annotators agree for the event. P_e is the summation of squared P_j , where P_j is the proportion of all assignments which were to the j -th category. In these equations, N is the number of Events, n is the number of annotators and k is the number of categories we have (Yes and No in our use case).

Based on the table for interpreting kappa values[12], we can say that the annotators have substantial agreement on the events annotated as being a relevant explanation for a water leak anomaly. Therefore, we can say that Scouter system was effective in selecting the most relevant events and provided a substantial help to contextualize anomalies related to water leaks.

However, we are still aiming at improving the evaluation of stored events based on the following 2 key points :

- Only 5 experts participated in this evaluation which is not sufficient regarding the scale of the use case. Therefore, we are preparing a new evaluation with dozens of field experts from various companies to assess more accurately Scouter.
- Data are collected over a short period related to 2017. We would like to extend the evaluation to the anomalies reported in 2015 and 2017 in order to test the full capabilities of Scouter.

6.3 Geo-profiling

The profiling module can be executed "offline" in our system. This means that it does not need to be integrated within the stream processing. In fact, the data required as input are mostly static: geographic data are rarely updated, and the network configuration does not change much over time. We performed a series of tests on several data sets given by the domain expert on both national and international levels.

Area	# Sensors	Available OSM data (Mo)	Profiling		
			Consumption ratio (ms)	POI (ms)	Region (ms)
P. Laval	2	5.4	270	25	605
V. Nouvelle	16	53.8	620	282	630
Hubies D.	1	5.8	190	13	30
Brezin	1	3.1	150	8	72
Guyancourt	2	4.2	161	13	50
Louveciennes	19	123.2	850	1118	1290
Hubies H.	13	37.15	740	163	180
Haut-Clagny	4	8.6	260	21	32
Garches	3	7.0	220	205	46
Gobert	3	15.4	201	36	105
Satory	5	32.5	292	103	215

Table 4: Performance table of the different methods of the algorithm

The performances of the reasoning module mainly depend on the size of the data extracted from Open Street Map: larger consumption sectors will have more data taking more time in the end. The profiling with polygons is the longest since it needs the extraction of both POI and polygons to be processed. On average, the computation of the consumption ratio is the fastest, since it doesn't need any extraction from Open Street Map.

Table 4 details the performance of our different methods for the use case of the region of Versailles (an area of 350.000 inhabitants in the suburb of Paris), which is composed of 11 consumption sectors. For each of these sectors, we indicate the number of flow sensors present on each sector, and the size of the data to extract from Open Street Map (both POI and polygons), then we indicate the computing time for each method. As we can see, the regions with the more data and the more sensors have the longest processing time: it corresponds to the biggest zones.

7 CONCLUSION

In this paper, we presented Scouter, a system that demonstrated its usefulness in the WAVES project for improving the contextualization of identified anomalies. The primary goal was to offer a generic system that can adapt to any Big Data platform whatever

the engine used and without losing the capability to process various types of data sources. To achieve such target, we chose an approach based on data connectors relying heavily on messaging queue system and we offer multiple NLP functionalities such as topic extraction and sentiment analysis.

Because of its easy-to-use Docker package, Scouter is already used in other prototypes at Atos and we are aiming to extend it with novel features such as ontology enrichment based on a dictionary of concepts and the identification of duplicate events coming from different data sources. Finally, we plan to improve the implementation by supporting various ontology formats (e.g. ttl, N3, RDF/XML, etc.) and adding new data sources to fit most use cases (e.g. traffic information, etc.).

Lessons Learned: During this work, the research team learned several lessons along multiple distinct dimensions. Instead of trying to adapt our implementation to heterogeneous technologies in the Big Data world, the best approach was to create a simple but powerful bridge that would make the integration seamless. The right choice was to go with a messaging queue system such as Apache Kafka that is widely used and known for its robustness. Moreover, we found out that few data sources with high quality content outperformed multiple data sources with medium to low quality content. Therefore, we decided to keep only 4 different categories and for each one, we selected the most suited source.

Since we are aiming at designing a tool adjustable to multiple use cases, going for an ontology-based approach seemed to be a win solution. Hence, we could give the possibility to every domain expert to use his own ontology with the specific concepts, properties and keywords that suit her needs. However, the key component for a successful implementation is to find the right models and the proper scores. Even though a lot of libraries were available for NLP functions, many of them lacked robustness and flexibility. Instead of investing much time on finding the fittest technologies, we are quite positive that we can highly improve the results by investing more time on ontology modeling and scoring.

Finally, we found out that the best way to remove complexity was to package the code into a user friendly web application. Therefore, instead of asking users to plug Scouter to their framework, they would just have to enter the location of the analysis, the specific data sources alongside with the proper domain ontology. As for the deployment part, using containerization technologies such as Docker tend to remove the burden of deployment and can be launched within a minute.

8 ACKNOWLEDGMENTS

This work has been supported by the WAVES project which is partially supported by the French FUI (Fonds Unique Interministériel) call #17. The WAVES consortium is composed of industrial partners Atos, Suez and Data publica, and academic partners ISEP and UPEM.

REFERENCES

- [1] H. Khrouf, B. Belabbess, L. Bihanic, G. Képéklian, and O. Curé, "WAVES: Big Data platform for real-time RDF stream processing," in *3rd Stream Reasoning workshop co-located with 15th International Semantic Web Conference, Kobe, Japan.*, pp. 37–48, 2016.
- [2] S. D. P. Adam Berger and V. D. Pietra, "A maximum entropy approach to natural language processing," *Computational Linguistics-MIT*, pp. 22–1, 1996.
- [3] S. de S Sirisuriya, "A comparative study on web scraping," *International Research Conference*, nov 2015.
- [4] J. Lovins, "Development of a stemming algorithm," *Mechanical Translation and Computational Linguistics*, 1968.

- [5] P. Domingos and M. Pazzani, "On the optimality of the simple bayesian classifier under zero-one loss," *Machine Learning*, 1997.
- [6] S. Ellouze, M. Jaoua, and H. Belguith, "Machine learning approach to evaluate multilingual summaries," in *Proceedings of the MultiLing 2017 Workshop on Summarization and Summary Evaluation Across Source Types and Genres*, April 2017.
- [7] C.-Y. Lin, "Rouge: A package for automatic evaluation of summaries," in *Proc. ACL workshop on Text Summarization Branches Out*, 2004.
- [8] D. J. O. H. A. Collomb, C. Costea and L. Brunie, "A study and comparison of sentiment analysis methods for reputation evaluation," tech. rep., Mar. 2014.
- [9] C. D. Manning, M. Surdeanu, J. Bauer, J. R. Finkel, S. Bethard, and D. McClosky, "The stanford corenlp natural language processing toolkit.," in *ACL (System Demonstrations)*, The Association for Computer Linguistics, 2014.
- [10] R. Socher, A. Perelygin, J. Wu, J. Chuang, C. D. Manning, A. Ng, and C. Potts, "Recursive deep models for semantic compositionality over a sentiment treebank," in *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, Association for Computational Linguistics, October 2013.
- [11] J. Fleiss *et al.*, "Measuring nominal scale agreement among many raters," *Psychological Bulletin*, vol. 76, no. 5, pp. 378–382, 1971.
- [12] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *Biometrics*, vol. 33, no. 1, pp. 159–174, 1977.

Finding Contrast Patterns for Mixed Streaming Data

Rohan Khade
George Mason University
Fairfax, VA, USA
rkhade@gmu.edu

Jessica Lin
George Mason University
Fairfax, VA, USA
jessica@gmu.edu

Nital Patel
Intel Corporation
Chandler, AZ, USA
nital.s.patel@intel.com

ABSTRACT

Contrast set mining identifies patterns in the data that can best distinguish between groups. Most of the existing work focuses on categorical and batch data, and they do not scale well for large datasets. In this work, we focus on finding contrast patterns for *mixed* (quantitative and categorical) and *streaming* data. We adapt a discretization methodology, Supervised Dynamic and Adaptive Discretization, to identify meaningful bin boundaries. We then use the discretization result to find contrast patterns on streaming data. In order to achieve this, we identify frequent items and then contrast them to a group of interest. To handle potential concept drift, we propose an update strategy to keep the frequent items relevant. In addition, our algorithm samples feature combinations based on their "sampling" score and user feedback to reduce the search space as well as retrieving more interesting patterns.

1 INTRODUCTION

As the amount of data being collected during the semiconductors manufacturing process of increases, the time required to analyze the data and provide relevant feedback also increases. There is a growing need to develop machine learning algorithms to deliver fast feedback to the engineers so that adjustments in the manufacturing line can be made in a timely manner. While the behavior of manufacturing data is often predictable, at times there exist anomalies such as a low yield for certain batch of products. To find the possible cause(s) of this low yield, one approach is to create a model and compare this batch to a normal batch. To achieve this, an engineer needs to know what to look for (the number of features and instances are large), and even after that he/she needs considerable amount of time for analysis. Our intent is to quickly and automatically learn complex patterns in the "population" data (known normal data) and then use these learned patterns to detect differences in the groups. We note here that our goal is data understanding rather than prediction. In prediction such as classification, the goal is to predict an outcome early on in the manufacturing process to reduce costs such as testing. The algorithm typically works like a black box where the user may not understand why the algorithm predicted an outcome. However, in this paper we design a feedback system in which the goal is to identify interpretable patterns that might provide the users helpful insights on potential causes of the differences. Another key difference is that for a prediction algorithms, the "classes" such as "good chips" and "bad chips" need to be known or available in advance in order to build a model. In our proposed work, we seek to detect patterns from a single group of data (the "population"), which will then be compared with incoming data to detect potential differences.

There are several issues that need to be addressed. First, the data may contain both categorical and continuous features, and the features may have high order of interaction between them. Multiple features could contribute to a contrast simultaneously. In this work, we leverage our recently proposed binning strategy, Supervised Dynamic and Adaptive Discretization (SDAD) [8, 9], to address the challenge of dataset containing continuous features. The discretization technique automatically determines the size and number of bins needed in order to find meaningful contrast patterns.

The second challenge we face is the massive data size. A large amount of data is generated daily. To obtain an understanding of the "normal" behavior of the data, we create a frequent itemset database using historical manufacturing data that are known to be normal. However, this data can change as recipes for different chips change. The frequent itemset database must adaptively learn new patterns, update existing ones and discard outdated ones. This is necessary because the representation of the "population" (historical) data needs to stay relevant to the current state of the database. With the updated representation of the population, we can quickly identify potential contrasts between that and a group of interest. If we try to build a database with only a small sample of the population data (e.g. data collected from one day), the data may be biased to the specific machine settings and may not be representative of the population. If the data is too large, e.g. a month worth of data, the data may contain outdated information that bias the results. Building an incremental database of the population itemsets over time can be more efficient and can potentially identify more meaningful patterns. Our algorithm is also easily parallelizable. While this is out of scope for this work, we refer interested readers to [8, 10] for various ways of achieving parallelization to find frequent itemsets.

Moreover, as data streams are monitored, the groups to be compared are not known in advance. Traditional contrast set mining algorithms require the contrasting groups to be known, and then find itemsets which differ significantly across groups. Consider two cases. First, suppose a tester detects a large number of faulty chips on a particular day, and we want to know if there is a difference between chips arriving at this tester and the others. One approach, taken by traditional algorithms, would be to make all the tester IDs as the 'group' label and then find patterns that strongly separate the tester of interest from the rest. As the second scenario, if an oven gives particularly high yield, we would like to know if we can improve the yield of the other ovens. Keeping the oven ID as the 'group' feature, the contrast set algorithm finds itemsets that are different between this oven and the rest. Since the "groups" are not known in advance, our proposed method identifies all frequent itemsets in the population. Once the "sample" of interest (e.g. the oven resulting in high yield) is found, the difference in supports between the population and sample is calculated. This led us to develop an algorithm which identifies frequent patterns[8]. Using frequent itemsets we can identify contrast sets between a sample (data of interest to an engineer) and the population.

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

The main contributions of the proposed work are as follows:

- (1) Given historic data (the “population”) and a current data sample, we propose an algorithm that can find contrast sets between any subset of the streaming data and the population quickly. Based on frequent itemsets mining, our approach avoids a lot of re-computation when new groups are identified and new contrasts are needed.
- (2) The proposed algorithm leverages user feedback to calculate a sampling score, which improves the patterns returned and reduces computation costs.
- (3) We propose an updating strategy that keeps discovered patterns current, discards outdated ones and adds emerging new patterns.

The following section discusses related work. Section 3 discusses the background on contrast sets and the search strategy adapted in this paper. In section 4, we describe an adaptive and dynamic discretization algorithm for quantitative data that improves upon existing discretization algorithms. We propose an updating strategy for finding contrasts on streaming data, as well as a strategy to use user feedback to compute the sampling probability. In Section 5, preliminary experimental results show that using our approach, we are able to discover the same contrasts that existing algorithms discover, but much more efficiently. Section 6 concludes the paper.

2 RELATED WORK

We discuss related work in frequent set mining and contrast set mining. However, due to the maturity of these areas, the discussion is far from being comprehensive. To the best of our knowledge, there is no other algorithm that has the exact same end goal as ours.

Contrast-set mining has been formally defined as “conjunctions of features and values that differ meaningfully in their distribution across groups” [1]. It can be viewed as a variant of association rule mining. While association rule mining discovers rules or patterns that describe or explain the current situation, contrast-set mining finds patterns that differentiate groups of data by identifying features and values (or conjunctions thereof) that differ meaningfully across them [1]. Knowing the features that characterize the discrepancies across various groups can help users understand the fundamental differences among them, and make independent decisions on those groups accordingly. Therefore, contrast-sets are often presented as sets of rules. The authors in [1] provide the following classic example: on comparing different education groups by asking “What are the differences between people with PhD and Bachelor degrees?” we might find that $P(\text{occupation} = \text{sales} \mid \text{PhD}) = 2.7$ while $P(\text{occupation} = \text{sales} \mid \text{Bachelor}) = 15.8$. For manufacturing fault analysis, we might have a query such as “Given two sets of operation sequences: G (“good batch”) and B (“bad batch”), what are the differences between them?” The authors in [1] proposed an algorithm, STUCCO (Searching and Testing for Understandable Consistent COntrasts), to find such contrast sets. STUCCO employs efficient search through the space of contrast sets based on another rule mining algorithm, Max-Miner [2]. To assess the meaningfulness of the difference in support values across groups, they use chi-square test on the null hypothesis that the support value is independent of group membership.

Another approach, proposed by [17], discovers that existing commercial rule-finding system, Magnum Opus [18], can successfully perform the contrast-set mining task. The authors conclude

that contrast-set mining is a special case of the more general rule-discovery task. A good survey on contrast sets, emerging patterns and subgroup discovery algorithms is provided in [13]. CIGAR [7], apart from using the pruning criterion from STUCCO, adds support, correlation and difference in correlation between the parent and child contrasts. The authors claim that itemsets with a support less than the minimum support may be uninteresting to the analysts. We adapt this assumption for the streaming part of our algorithm. The authors also claim that if the itemset have low correlation, we may find spurious contrasts due to outliers. The authors in [4] use user feedback to sample the patterns and improve the patterns displayed to the user; however, they do not explicitly handle continuous and streaming data.

In [13] we see that Contrast Set Mining, Emerging Pattern Mining and Subgroup Discovery are compatible and hence techniques developed in one domain can potentially be used in another. There is considerably more work done for finding subgroups in numerical domains. The algorithms presented in [11, 15] finds bins for subgroups for mixed data and is implemented in an open source tool Cortana. These techniques usually use an initial discretization method and then merge spaces based on an interest measure. An interesting algorithm described in [6], finds bins for continuous attributes for the problem of subgroup discovery. The algorithm uses optimistic estimates and horizontal pruning to prune the search space. This heavily relies on pruning based on finding the top-k subgroups. Finding all initial split points (exhaustive search in [6]) is expensive but if the initial partitions are not exhaustive but rather frequency or entropy based, the algorithm may miss interesting patterns that occur lower down the tree due to multivariate interactions. The techniques mentioned above are developed for static databases, however the pruning techniques described may be helpful if the groups are known in advance (which does not apply for our application). The current trend of recent algorithms tend to use sampling to improve efficiency and quality of patterns [3–5]. We use some of the sampling techniques explained in these papers and incorporate user feedback to enhance it. This helps improve efficiency and potentially display more meaningful results to the user.

Although our main goal is contrast set mining, we tackle the streaming part of our work using frequent itemsets mining. Frequent itemset mining is typically the first step of association rule mining. This is also where the main computational complexity issues occur. Most existing work in frequent set mining focuses on batch processing and improving the efficiency of the algorithm. However, in many real world applications, new data is continuously generated, e.g. manufacturing processes, sensors etc. General frequent itemset mining algorithms require multiple passes over the database. However, this is not possible in a streaming scenario since data come in high volumes. If we miss a pattern, it may not be possible to go back and check the past data. Since speed is an important issue, some trade-offs in accuracy may be needed. In addition, the space required to store information for future runs and current knowledge may be large.

In general, many streaming algorithms adapt the window model [14]. In landmark window, the goal is to find itemsets between a fixed start point s and current time t . The other option is a sliding window where we are only interested in itemsets found in a time frame. For example, if the window width is w , the goal is to find itemsets in the time $[t - w + 1, t]$. In many cases newer data are more important than older data, in which case a damped window model assigns higher weights to more recent data by defining a decay rate. Time Tilted windows are

used to find frequent itemsets over a set of windows. Importance is given to newer data and hence granularity is adjusted as data arrive. Since our goal is to find the general behavior of the data, we use a sliding window strategy, assigning equal weights to all the data.

There are two types of streaming algorithms to find streaming frequent itemsets, true positive algorithms and true negative algorithms. A true positive algorithm does not allow any frequent itemset to be missed. The seminal work by Manku and Motwani [12], called *lossy counting*, uses a user-defined parameter ϵ , and the result contains no itemsets with a support lower than $\delta - \epsilon$. The main disadvantage of this approach is that the number of itemsets increases exponentially to guarantee the lower bound. On the other hand, true negative algorithms [19] do not find false positives, but have a higher probability of finding true frequent itemsets. The main limitations of these algorithms are that it is difficult to implement them in parallel; they cannot handle continuous features; and the space requirement may be too high if the data are very large. In this work we use a true negative algorithm; however, as will be discussed later, we can estimate the support of itemsets are missed in the stream of data.

3 BACKGROUND

3.1 Contrast Sets

Bay [1] formally defines contrast set mining as follows. Let $A = \{a_1, a_2, \dots, a_j\}$ be a set of all attribute values for the entire dataset. Let C be a set of all combinations of A . An *itemset* c is a subset of C . Let $G = \{g_1, g_2, \dots, g_m\}$ be a set of groups. Each instance belongs to only one group. If $|g_i|$ is the number of instances in group i and c_i is the itemset of interest in group i , then **support** s_{ic} of an itemset c in group i is the number of instances in g_i that contain c :

$$s_{ic} = \frac{\text{count}(c_i)}{|g_i|} \quad (1)$$

An itemset c is **large** between two groups i and j if

$$\text{abs}(s_{ic} - s_{jc}) > \delta \quad (2)$$

and **significant** if

$$\chi_{ij}^2(c) < \alpha \quad (3)$$

where α and δ are user-defined parameters.

An itemset is considered a contrast if it is large and significant.

Three pruning strategies are used to reduce the search space. (1) An itemset is pruned if it does not have a support over δ in any group (*minimum deviation size* pruning). (2) If its expected occurrence is less than 5 since statistical tests are not significant at that level. (3) By calculating the upper bound value of χ for the itemset's children. To reduce the number of false positives, the value of α is adjusted according to Bonferroni's adjustment explained in [1]. We note here that the pruning strategies used by STUCCO cannot be used in our application since we do not know the groups in advance. However, as will be seen later, we only find itemsets with support greater than δ for the streaming data, and if the group of interest has a support greater than δ , we estimate its support from the population sample.

3.2 Search Strategy

To find combinations of attributes, we need a search strategy. To this end, we adapt the OPUS [16] search tree for our algorithm. However, some modifications are needed since the original tree is designed for categorical attributes. Figure 1 shows the modified

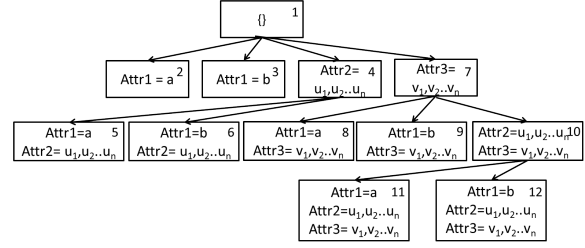


Figure 1: OPUS Search Tree for Mixed Data

OPUS tree so that it is compatible with continuous attributes and attribute-value pairs instead of directly handling itemsets. Consider an example with one categorical attribute *Attr1* having possible values a and b , and two continuous attributes *Attr2* and *Attr3* having values $u_1, u_2 \dots u_n$ and $v_1, v_2 \dots v_n$ respectively where n is the number of instances. The order in which the nodes are traversed is indicated by the number in each node. In this example, node 2 with itemset $\{a\}$ is first explored. If it can be asserted that node 2 can be pruned, then we do not need to explore nodes 5, 8 and 11. It should be noted that the traversal order is the reverse of depth-first search. The reason that we use this approach instead of a depth-first approach is to maximize pruning. Although breadth-first search can also be used, we opt for OPUS since the storage overhead is less at each level.

3.3 Streaming Frequent Itemsets / Patterns

Frequent Itemsets (FI's) are itemsets which are present in the dataset with a frequency greater than a user-defined threshold. The formal definition follows closely to that of contrast sets but notice the subtle differences. Using the definition of c from the earlier paragraph, let T be a dataset of transactions. If $|T|$ is the number of rows in the dataset then **support** s of an itemset c is the number of times c occurs in T :

$$s_c = \frac{\text{count}(c)}{|T|} \quad (4)$$

A streaming dataset τ is a sequence of indefinite number of datasets of transactions $\{T_1, T_2, T_3, \dots\}$. Let $T_{k,l} = \{T_k, T_{k+1}, \dots, T_l\}$ where $k < l$. The support s of an itemset c in a time window $[k, l]$ is:

$$s_{c[k,l]} = \frac{\text{count}(c_{[k,l]})}{|T_{k,l}|} \quad (5)$$

4 STREAMING CONTRAST SETS

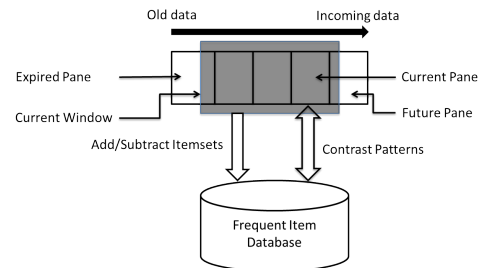


Figure 2: Sliding window for Data Streams

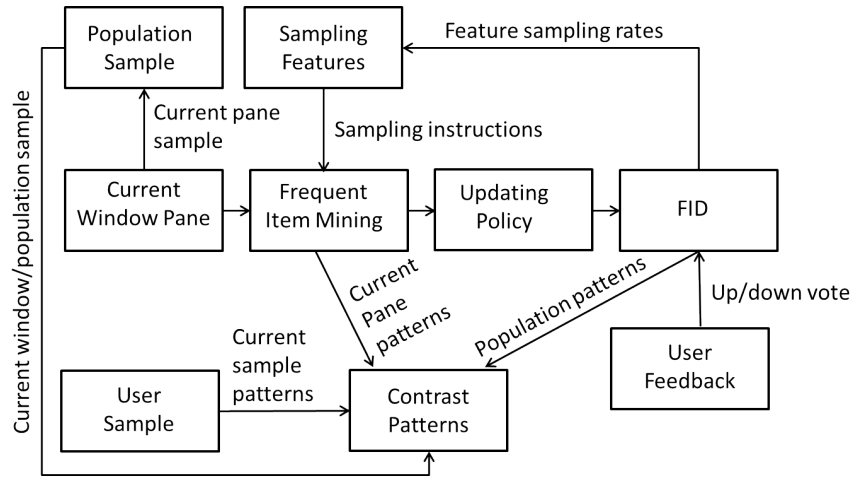


Figure 3: Data flow

We start our discussion by explaining some of the basic terminology used in this section. As shown in Figure 2, data continuously arrive in a data stream. Each block of data is called a **pane**, and a group of panes that are relevant is called a **window**. In Figure 2, suppose a pane represents a day’s worth of data, then the window size, $\lambda=3$, represents 3 days. Itemsets that are no longer frequent in this window can be discarded.

Figure 3 provides a high-level view of our proposed algorithm. In the next few subsections we discuss what is computed in each block and what information is communicated by each arrow. Simply stated, we pre-compute the frequent itemsets for the current window pane from the population data. This is a time-consuming task, given the massive size of the database. The frequent itemsets discovered from the population data will be saved in a **Frequent Itemset Database** (FID from here on). We note here that this database is not static; it is updated with the arrival of new data as shown in the figure. A data structure such as a prefix tree (like Figure 1) can store this information. When a new **window pane** arrives, frequent itemsets from the current pane will be computed, using techniques explained in a later section. These frequent itemsets are then compared with the population frequent itemsets computed previously to obtain potential **Contrast Patterns**. If the new pane does not conform to the patterns found in the previous panes, the engineers can be notified in a timely manner.

The user can also provide a sample dataset that he or she wishes to compare against the population data, or generate the sample from the archived data on the fly as seen in the **User Sample** block in Figure 3. The frequent itemsets from the sample data are extracted and they will be compared against the itemsets retrieved from the FID for the population data to find **Contrast Patterns**.

Sometimes a pattern that is found in the Current Window Pane or the User Sample may not be in the FID since its support in the population may be below the threshold. To overcome this, we keep a **Population Sample** if the algorithm encounters such a situation. The supports of the population are then estimated from this sample.

The intuition behind our algorithm is that the patterns for the population data represent the “normal” behavior of the population, and comparing the set of patterns from sample data against this “normal prototype” would reveal any differences between the

two datasets. Such discovery could potentially identify anomalies in the current or sample data, and shed some light on the causes of the anomalies (by examining the rule differences).

After finding the frequent itemsets for the current pane, the algorithm updates the FID with the new frequent itemsets found using our **Updating Policy**. As new data come in, the distribution and relationships may change among the features. To keep the FID relevant, we need to remove patterns that are no longer relevant and add emerging patterns. If the algorithm finds a new pattern, we delay adding it to the database until we keep finding the same pattern in multiple iterations and if an existing pattern is no longer significant, it is phased out. The user can keep track of these changes in a timely manner.

The **User Feedback** block lets the user decide if a pattern is subjectively interesting. This in turn is fed to the **Sampling Features** block which improves the efficiency and displays more interesting patterns.

In addition to offering the ability to compute the contrast-set in real-time given the sample data, another advantage that our approach offers is that the contrasting patterns are not restricted to having only the values of a feature representing group membership. We will provide more detail on the algorithm in the next section.

4.1 Finding Frequent Itemsets for Current Window Pane

While traversing the OPUS search tree explained in the previous section, if we reach a node with only categorical features, finding frequent itemsets for the current window pane for categorical variables is straightforward, i.e., we just need the support count of the itemsets in the current pane. Therefore, we will concentrate on itemsets that contain continuous features. To this end, we use Supervised Dynamic and Adaptive Discretization for frequent itemsets (SDAD-FI) [8]. We note that any frequent itemset mining algorithm that can handle mixed attributes can be applied here; however, we opt for SDAD-FI since it can handle datasets with mixed attributes and is parallelizable. We also show in [8, 9] that SDAD is able to find better quality bins for frequent itemset mining and association rule mining in a reasonable amount of time compared to other state of the art binning and rule mining algorithms. As opposed to standard discretization techniques such as equi-width or equi-frequency, SDAD-FI does not need to

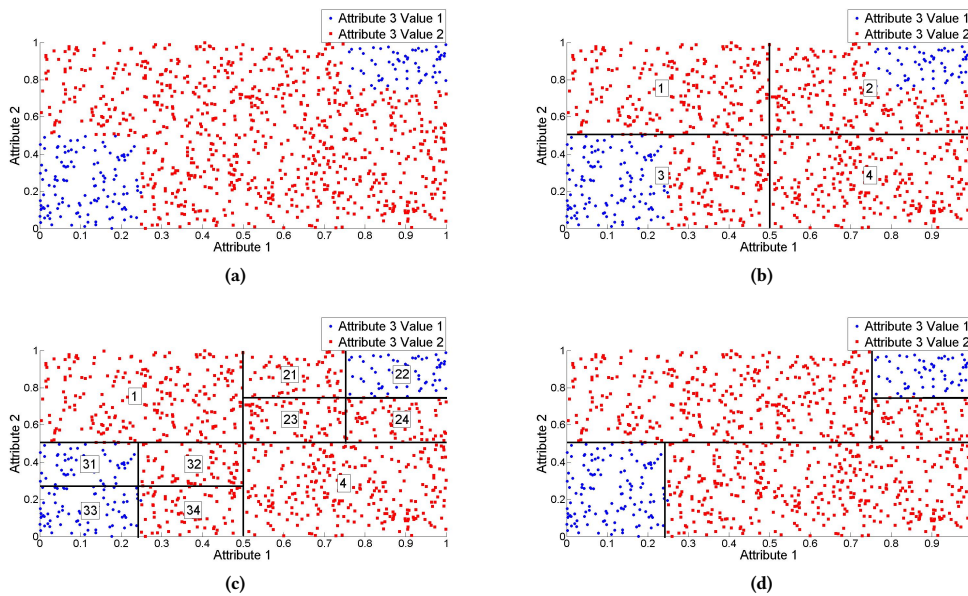


Figure 4: (a) Simulated dataset with 3 features (b) After first partition between feature 1, 2 and 3 (c) After second partition between feature 1, 2 and 3 (d) After merging

know the number of bins to partition in advance, nor the sizes of the bins. It tries to find the "best" bins with respect to the distribution and relationships between features.

The main idea of SDAD-FI is to first employ a top-down approach where the algorithm iteratively partitions the space between features and calculates an interest measure with the goal to obtain regions that satisfy a user-defined homogeneous threshold. We use the interest measure *purity* as defined in [8]. The intuition behind purity is to find the most homogeneous space of the current set of features. We will describe the algorithm with an example. Consider Figure 4a. This dataset contains three features: Attributes 1, 2 and 3. Attributes 1 and 2 are continuous and Attribute 3 is categorical with 2 values indicated by the color and shape of the markers in the scatter plot. In Figure 4b, SDAD-FI divides the continuous features into 4 spaces. Space 1 and 4 (numbers shown in figure) are homogeneous, i.e. it contains only one categorical feature value. However, space 2 and 3 are not. SDAD-FI again divides those spaces as shown in Figure 4c and now finds pure spaces. Since dividing pure spaces do not find "better" bins, the algorithm stops, and this concludes the top-down process.

The next step is to merge contiguous and similar spaces in a bottom-up fashion. Similar spaces are those that are not significantly different according to the χ -squared test. SDAD-FI starts with the smallest contiguous spaces in terms of *n-volume* (area in this case). It first merges space 21 and 23, then 31 and 33, and then 32 and 34. It further merges the combined space of 21 and 23 into space 1, and 32 and 34 into space 4. The final bins are shown in Figure 4d.

Obviously, the example shown is an ideal case. In real world datasets, we usually do not find such clear cut boundaries. However, this example illustrates the flow of the algorithm. In general, SDAD-FI is a greedy algorithm and continuous features are split until splitting does not result in statistically significant differences between the spaces (instead of looking for pure bins). The

example discusses a case in which categorical and continuous features co-exist in the itemset. However, SDAD-FI also works when there are only continuous features in the itemset. In this case, the splitting and merging operations are as explained above, with the modification that the interest measure is now the correlation between the continuous features. See [8] for more details.

4.2 Updating Policy and FID

If an itemset contains only categorical values, the Frequent Itemset Database (FID) is updated if a difference in the supports of a Frequent Itemset (FI) is detected between the current pane and the FID using the following update policy. Consider the example in Figure 2. Let the minimum support be δ . If the size of the window is λ , then in Figure 2 we have used $\lambda=3$. Now suppose there is an FI in the FID with support s_{FID} , and the same FI is found in the current pane with support $s_{current}$, we calculate the approximate support of the itemset $s_{expired}$ in the expired pane from the population sample (explained later). The support of the FI in the FID can then be updated using a simple weighted average technique.

$$s_{new} = s_{FID} + \frac{s_{current}}{\lambda} - \frac{s_{expired}}{\lambda} \quad (6)$$

If a new FI is found in the current pane with a support greater than δ , but it is not in the FID, then applying the updating strategy explained above would not work since $(\lambda-1)*s_{FID}/\lambda$ would be zero and $s_{current}/\lambda$ may not be big enough to be greater than δ to be added to the FID. For example, suppose $\delta = 0.1$. If in the current pane we find an itemset with support of 0.9, and if $\lambda = 10$, then s_{new} would be 0.09. As a result, the FI would not be able to enter the FID even if it is found consistently after a certain point. To overcome this, we approximate the support (s_{FID} in the above formula) from the **population sample** and keep a temporary database of new FI's.

The itemset is stored in a temporary database until the itemset attains a support greater than the minimum support. After this

point, we say the itemset is indeed frequent and add it to the FID, as well as deleting it from the temporary database. There is a possibility that the current itemset is an anomaly and is not truly frequent. In this case, the itemset needs to be purged from the temporary database. Let the number of panes, starting from the first pane the itemset was found to be frequent, be n . The algorithm keeps track of the actual support (s_{act_supp}) as soon as the itemset was discovered as frequent, and it is calculated by

$$s_{act_supp} = \frac{(n-1) * s_{act_supp} + s_{current}}{n} \quad (7)$$

As long as s_{act_supp} is above the minimum support threshold, the algorithm keeps the itemset in the temporary database.

We use this approach to add or purge itemsets because our goal is to maintain a consistent FID which is representative of the population. For example, if an FI is in only one pane, it may be an outlier and not representative of the population. However, if an itemset appears regularly in multiple panes, but is absent for some reason for a short period of time, the algorithm should not purge it out. We note here that keeping the population sample to estimate the support of unknown itemsets reduces the number of frequent patterns needed to be stored like in [11, 19], resulting in less computation. Although the storage overhead is higher to keep the population sample, it serves another purpose. It also helps when the user wants to find contrast patterns on the fly, and the itemset is not present in the FID.

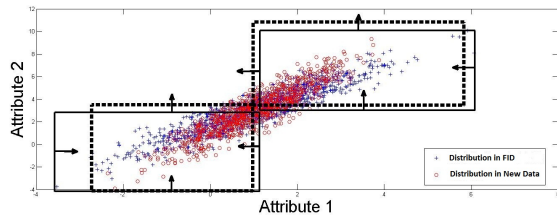


Figure 5: Bin movement

The method explained can be used for frequent itemsets containing only categorical features. Updating FI's with continuous features needs a different approach. Binning is an expensive operation and, for each pane, it is unlikely to have exactly the same bin boundaries even if the distributions are statistically the same because of noise in the data. Instead of calling SDAD-FI at every pane, first, the continuous variables of the current pane are binned by re-using the bin boundaries of the current FID. If the supports between the FID and current pane are not significantly different using the χ -squared test, the bins in the FID are kept unchanged. However, if that is not the case, we discretize the itemset in the current pane by running SDAD-FI. At this stage, we check if there is a slight concept drift or if a completely new relationship is learned between the features. The algorithm conducts a test to check if the new bin boundaries overlap with the current boundaries, and if so, by how much. To do this, there has to be an overlap in every axis. For example, suppose the current itemset has 2 continuous features (the bin boundaries in this case form rectangles, see Figure 5), x_1 and x_2 are the vectors of the x coordinates, and y_1 and y_2 vectors of the y coordinates of the 2 rectangles, then there is an overlap between the rectangles

if $((\min_{x_1} \leq \min_{x_2} \ \&\& \ \min_{x_2} \leq \max_{x_1}) \ || \ (\min_{x_2} \leq \min_{x_1} \ \&\& \ \min_{x_1} \leq \max_{x_2})) \ \&\&$

$((\min_{y_1} \leq \min_{y_2} \ \&\& \ \min_{y_2} \leq \max_{y_1}) \ || \ (\min_{y_2} \leq \min_{y_1} \ \&\& \ \min_{y_1} \leq \max_{y_2}))$

In general, to decide if there is significant overlap, the ratio of the n-volume of the overlapping space to the minimum n-volume of the original space and the new space is calculated. This is a user-defined value (we use 0.8 in our experiments). If there is significant overlap, the algorithm moves the current boundaries towards the new boundaries as shown in Figure 5. The figure shows the distribution of the data points in the FID and the current pane represented by the blue and red points respectively. The bold rectangles represent the bin boundaries of the FID, and the dotted lines are the new bins boundaries found. The arrows indicate the directions of movement of the boundaries of the new bins. The movement is again calculated by using a weighted average method, with the weight determined by the number of days the original pattern was in the FID. For example, if the number of days the pattern was in the database is n and the perpendicular distance to the new hyperplane is x , then the actual distance the current hyperplane moves is x/n . We note here that each plane (in this case line segment) moves independently of each other. The number of days in the FID is then incremented by 1, and the support is approximated using Equation 6 by calculating the supports of the new bins. More specifically, we add the support of the current pane and subtract the support of the expired pane calculated from the population sample.

If the new boundaries of the current pane are significantly different from the original boundaries, the new itemsets with the new bins are stored in the temporary database. In the next panes, the algorithm checks if the data conforms to these new bins or if this behavior was just an anomaly. This procedure is similar for itemsets containing just categorical features.

4.3 Sampling Patterns and User Feedback

During the manufacturing process, the interaction between features are not expected to change drastically. We can exploit this predictability by using a sampling method to decide which itemset should be checked in each pane. Sampling helps reduce the search space for each run (and potentially display more interesting contrasts to the user). This is achieved by calculating a "sampling" score for each itemset in the database. We identified three factors that should contribute to this score.

- (1) Number of days in database
- (2) Change in support
- (3) User feedback

4.3.1 *Number of days in database.* In the manufacture of semi-conductors, we do not expect the interaction between the features to change drastically on a daily basis once the recipe for a chip is fixed. However, concept drift is possible due to aging of machines or inherent properties of the process. This is potentially interesting for our algorithm. Patterns that have been found consistently tend to be stable and show very little variability on the daily basis and hence it is less useful to calculate its interest measure in every pane. Patterns that are newly discovered (i.e. the patterns in the temporary database) tend to have more variability and need to be sampled at a higher rate. If n is the number of days since a pattern is found, and λ is the window size, the $score_{nd}$ is calculated by

$$score_{nd} = 1 - \frac{n}{\lambda} \quad (8)$$

This linear equation produces results ranging from 0 to 1. The score is inversely proportional to the number of days the pattern is present in the database.

4.3.2 Change in support. If a pattern is newly discovered, not found any more or has a high difference in support from what was found previously, the algorithm should sample this pattern at a higher rate. We calculate the score_s as

$$score_s = \frac{abs(s_{FID} - s_{current})}{max(s_{FID}, s_{current})} \quad (9)$$

This score also ranges from 0 to 1, with 1 meaning that the pattern is new or is not found in the current window. It is also proportional to the change in support.

4.3.3 User feedback. Pattern mining algorithms tend to find spurious patterns due to the bias of the interest measures or due to the domain knowledge of the user (some patterns may be obvious or inherently not interesting). To overcome this bias, the proposed algorithm lets the user up vote or down vote a pattern displayed. Doing so allows the algorithm to learn more important combinations of features over time. With the user voting information, the algorithm will sample these interesting combinations at a higher rate so as to be less likely to miss changes of these features. Let v_{max} be the maximum number of up votes a pattern can receive, v_{min} be the maximum number of down votes a pattern can receive, and v_{up} and v_{down} be the numbers of up and down votes a pattern actually receives from the users, respectively, then

$$score_u = \frac{v_{min} + (v_{up} - v_{down})}{v_{max} + v_{min}} \quad (10)$$

Different users may have different preferences on the types of patterns that are interesting. Hence, we cannot remove the pattern from the FID when a user down votes a pattern. Rather, the algorithm needs to keep track of the votes and not display such pattern again to the same user.

If \mathbf{w} and \mathbf{score} are the vectors representing the weight for each score and score calculated, then

$$score_{total} = \mathbf{w} \cdot \mathbf{score} \quad (11)$$

Once the score is calculated, we use a logistic function to calculate the probability ϕ of sampling the itemset

$$\phi = A + \frac{K - A}{1 + e^{score_{total} - mean(score)}} \quad (12)$$

where A and K are the lower and upper bound of the probability of sampling. The expected number of panes after which a pattern's interest measures are calculated is $1/\phi$

4.4 Finding Contrast Patterns and Population Sample

Given a sample, we wish to compare against the population. To this end, we explore the search space of the sample using OPUS search. At each node, the algorithm checks the FID to see if it is an itemset that has been previously found. If it is, and the itemset has only categorical values, then comparing them is straightforward. Let λ be the window size and σ be the average number of tuples present in each pane, then the total number of itemsets (needed for χ -squared calculation) is given by:

$$N = \lambda * \sigma \quad (13)$$

Furthermore, let s_c be the support of itemset c in the database, the number of times c is present in the database is:

$$N_c = s_c * N \quad (14)$$

After finding the counts of an itemset, the algorithm can proceed to check whether the itemset is large and significant as explained earlier in the background section. Notice here that we are only interested in itemsets that are more frequent in the sample than the population.

If the itemset is present in FID and contains at least one continuous feature, the algorithm discretizes the continuous features using the bin boundaries previously found and recorded in the FID for these same features. Once discretized, the algorithm can treat the features as categorical features, and can continue as explained above. If the itemset is not large and significant, this suggests that the distribution and relationship between features is not different in the sample.

A problem arises if the distribution or relationship between features change, or if the itemset is not frequent in the population but found in the sample (since that information is potentially lost). To overcome this, we keep a population sample \mathbf{d} that fits in memory. At each window pane, the algorithm randomly samples k instances of the current transactions. If the algorithm has not finished running for at least (λ) panes, k instances are added to the end of \mathbf{d} . Otherwise, when a new window pane arrives, the algorithm removes the first k instances and adds the current sample to the end. Therefore, the maximum number of tuples in \mathbf{d} is $k * \lambda$. Now, if an itemset which is frequent in the current window pane is not found in the FID, the algorithm can go to \mathbf{d} and calculate the approximate support. Again, if the itemset contains only categorical features, this is straightforward. If the itemset contains continuous features, the algorithm discretizes the samples itemset using SDAD-FI, and uses the bin boundaries to calculate the support of those bins from \mathbf{d} .

5 EXPERIMENTAL EVALUATION

Experimental Setup: For all the experiments, we keep *min pr* difference 0.1 (for merge step in SDAD-FI), $\lambda = 5$, $\alpha = 0.05$ and $\delta = 0.1$ (same as [1]). The datasets used to quantitatively compare the performance of the algorithms are shown in Table 1. Each dataset is divided into 5 random partitions. The parameters A and K are set to 0.2 and 0.8 respectively, which means that each itemset has a minimum of 0.2 and maximum of 0.8 probability to be tested in each pane. For user feedback, we randomly up vote or down vote an itemset with probability 0.1. The population sample is capped at 10% of each partition. A minimum support of 0.05 is used for experiments in Table 3.

5.1 Public Datasets

These are static datasets. By creating random partitions, we should find similar distributions of the features in each partition. We are not aware of any algorithm that works similarly to ours, or finds contrast patterns for mixed and streaming data. Hence, the goal of these experiments is to verify that the proposed approach is able to find contrast patterns comparable to the actual contrasts found if the entire static dataset were used. We compare our algorithm with STUCCO on the entire dataset (since it is a batch algorithm), by binning the continuous attributes using the bins found by SDAD-FI, and then running STUCCO on the discretized data.

Table 1: Public Datasets

Dataset	Sample	No. of instance Dataset/ Sample	No. of Features/ Contin- uous Features
Adult	Doctorate	48842/594	13/5
Spambase	Spam	4601/1813	57/57
Shuttle	High	54489/8903	9/9
Credit Card	Yes	29998/6635	24/23
Census Income	Above 50K	199523/12382	39/11

Table 2: Average Time in seconds to find Frequent Patterns in each Pane for Public Datasets

Dataset	Support=0.05	Support=0.1	Support =0.2	Support=0.3
Adult	17.36	16.49	10.37	10.69
Spambase	27.18	23.88	24.11	26.09
Shuttle	9.13	11.02	7.68	11.03
Credit Card	29.41	28.08	23.7	32.56
Census Income	232.25	189.77	154.98	157.08

Table 3: Quantitative Analysis of Contrast Sets for Public Datasets

Dataset	Time to find Contrast us- ing SDAD FI (seconds)	Time to find Contrast on entire dataset (seconds)	True Positive	False Positive	False Negat- ive
Adult	3.09	65.02	40	14	26
Spambase	121.59	952.87	27	2	2
Shuttle	4.25	62.33	9	0	0
Credit Card	45.25	482.74	8	0	0
Census Income	148.72	1654.85	228	0	24

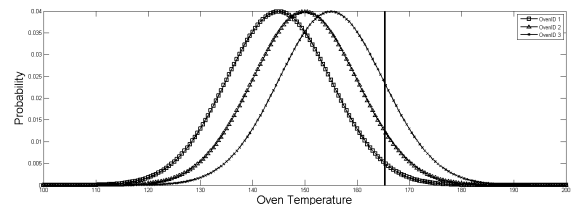
The 5 partitions are fed into the algorithm sequentially and the FID is updated. Finally, the "User Sample" (shown in the table as sample of interest) is compared to the FID.

In Table 2, we report the average time it takes to find frequent patterns for each pane for different minimum supports. The results indicate that not only the minimum support affects the running time, but the number of items needed to be updated in the FID also matters. We notice that for some datasets, even increasing the minimum support increases the running time, which is counter intuitive. We can compare the first column (Support = 0.05) of this table with running SDAD-FI on the entire dataset in Table 3. The total running time is actually greater (average time multiplied by 5) because of the time require for updating and computation, however, we see that once the algorithm pre-computes the frequent itemsets, it can find contrast patterns quickly (see first 2 columns of Table 3). We also compare the difference between the contrasts found using the windowing procedure and the actual difference in support by showing the true positives, false positives and false negatives.

The results show that our approach finds contrasts comparable to contrasts found when using the entire dataset. It also shows that the time it takes to find the contrasts is much faster at run time than using the entire dataset.

5.2 Simulated Dataset

We simulated a dataset which is a simple but typical scenario in our application, semiconductor manufacturing. This dataset gives an overall idea of what we are trying to achieve. The dataset contains 4 features: the date, the oven id, temperature and a

**Figure 6: Simulated dataset temperature data distribution****Table 4: Frequent-1 Items found in Population by SDAD-FI for Simulated Streaming Datasets**

Frequent Item	Support
Response0	0.93
Response1	0.07
Temperature <= 150.08	0.5
Temperature > 150.08	0.5
Oven2	0.33
Oven1	0.33
Oven3	0.33

Response which indicates a pass or fail for a particular test. Each instance represents a chip and has a unique id. In the simulated data, we have 3 oven ids, and for each oven the temperature is controlled individually and follows a normal distribution. We simulate this for 10 batches, each having 600 instances, 200 for each oven. The probability distribution of the data is shown in

Table 5: Contrast Sets for Simulated Streaming Dataset

Serial Number	Contrast Set	Support for Population	Support for Sample
1	<i>Temperature</i> <= 150.08	0.5	0.31
2	<i>Temperature</i> > 150.08	0.5	0.69
3	<i>Temperature</i> <= 150.08	0.5	0
4	<i>Temperature</i> > 150.08	0.5	1
5	Oven3	0.33	0.58
6	Oven1	0.33	0.11
7	<i>Temperature</i> <= 150.08	0.5	0.45
8	<i>Temperature</i> > 150.08	0.5	0.55
9	Response 1	0.07	0.11
10	Response 0	0.93	0.89

Table 6: Contrast Sets for Semiconductor Manufacturing Streaming Dataset

Serial Number	Contrast Set	Support for Sample	Support for Population
1	<i>Tester</i> – KAWM	0.51	0.27
2	<i>Tester</i> – KAWM and Burn in Flag – 80	0.51	0.25
3	<i>Unit Location at CAM</i> – JVFT and <i>Entity at Deflux</i> – VAKM	0.55	0.33
4	<i>CAM Location</i> – 70	0.50	0.33
5	10.6435 <= CAM PEAK TEMP STD <= Inf	0.55	0.42
6	91.5456 <= CAM DIE TIME ABOVE AVG <= Inf and <i>PRODUCT</i> – YBDF	0.47	0.33

Figure 6. Ovens with IDs 1, 2 and 3 have mean temperatures 145, 150 and 155 respectively, and each has a standard deviation of 10. Instances that failed the test have Response = 1, while those that passed have Response = 0. The probability of failure follows a sigmoid distribution where the mean of the sigmoid is 165, as shown in Figure 6. The minimum probability of failure is 0.05, and maximum is 0.9. Hence the probability of failure for an instance is given by

$$\phi = 0.05 + \frac{0.9 - 0.05}{1 + e^{temperature-165}} \quad (15)$$

Using this scenario, we try to answer 3 questions: (1) We notice that the oven with *OvenID* = 1 yields a high number of chips that passed the test. What is the difference between the daily runs of *OvenID* = 1 and the population? (2) On similar grounds as (1) but globally, given a sample of chips that have failed the test, can we find meaningful differences? (3) If, on a particular day, we change the temperature, can we provide a timely feedback on how it is affecting the Response? We note here that to answer the questions above, a traditional contrast set algorithm needs to be run thrice on the entire data, which is time consuming since the data is large in a real world scenario. Table 4 shows the Frequent 1-Items found.

For the first scenario, we took a sample that contains only *OvenID* = 1. In Table 5, the first 2 contrasts are the ones that are significant at *level* = 1. As the contrasts show, *OvenID*=1 is operating at a lower temperature.

In the second scenario, looking at contrasts 3 to 6 we notice that the bad samples are baking at a higher temperature, and *Oven3* is failing much more than average. Looking at the contrasts, the analyst can recommend operating the oven at a lower temperature.

In the third scenario, we increased the mean of each oven by 2, keeping the standard deviation at 10 for a particular batch. When the algorithm runs daily, it finds that this batch has a higher

rate of failure than usual and can notify the analyst as soon as something goes wrong. From the contrast, we see again that the increase in temperature might be the cause of the high rate of failure.

All the contrasts shown find very simple contrasts but provide useful feedback to the analyst. Some of the frequent 2-itemsets found are *Temperature* <= 165.02 and *Response* = 1, *Temperature* > 165.02 and *Response* = 0. In a real life scenario, contrasts can be more complex, and bin boundaries can be fuzzy. The frequent itemset mining algorithm is capable of finding these rules [8]. In initial runs, the date feature was listed as strong contrasts. This was down voted by the user and soon the algorithm learned it was not an important feature, hence reducing its sample rate to the minimum.

5.3 Semiconductor Manufacturing Dataset

We conducted experiments on real production data of a certain product group. Note that the data was cleaned and modified to remove any proprietary information, although relationships between attributes remain unchanged. The goal of this experiment is to see whether one can quickly identify the root cause of why parts are failing a specific test. The dataset consists of 15 days worth of data. Each day consists of 3,000 to 20,000 instances of individual units, and each instance has 174 features. We built an FID, and we then contrasted the failing units to this database. The contrasts found are shown in Table 6.

The contrasts found shed some light on the potential causes of the problems. In Table 6 contrasts 1 and 2 show that most of the failed parts are routed through the same tester and had been flagged for burn in. Contrasts 3 and 4 show there was a higher likelihood a particular chip attach tool was used (*CAM Location*) and location of the unit in the tray (towards the back of the oven) was used. This may explain contrasts 4 and 5 which state that the chip attach reflow temperatures were on the higher end of the

spectrum. So in summary, this situation exhibits a marginality between the chip attach process, the presence of burn-in and potential marginality at that specific test equipment. With this information, engineers can make changes at these particular stages of the manufacturing line, which may in turn improve the process yield.

As mentioned earlier, we do not expect the distribution and relationships between the features to change a lot on daily basis, and hence the sampling procedure will help improve the running time. The first pane of the algorithm is run without any sampling. Since the FID is empty, all bin boundaries have to be calculated. The average time for the first run is 2456 seconds. After the first pane, the average time for a pane to update the FID with the sampling procedure explained earlier is 182 seconds, and without sampling it is 515 seconds. This improvement in the running time is significant, but the contrasts found do not differ significantly. We calculated the difference in supports between the population and sample for the top 25 contrasts in the experiment without sampling. The corresponding contrasts from the experiment with sampling were extracted. As expected, the continuous attribute bins were slightly shifted when compared between the 2 experiments. The absolute average difference of differences of supports for population and sample between the 2 experiments is 0.017 with a standard deviation of 0.041. Their distributions are not significantly different at $\alpha = 0.05$ according to the Wilcoxon signed rank test.

6 CONCLUSION

In this paper, we propose a method to find Contrast sets in mixed and streaming data by extending a well-studied approach. Using a binning strategy that automatically finds the size and number of bins for the continuous features, we are able to find meaningful contrasts even when higher order interactions occur. An updating strategy was discussed to keep patterns relevant. To find contrast patterns, we first find the frequent patterns for the population, which is then contrasted to the sample of interest. This algorithm does not need the group information in advance. The main motivation to find these types of contrast patterns in streaming data is to provide timely feedback to analyst during the semiconductor manufacturing process. The next step for our work is to improve the parallelization capacity for our algorithm to run on even larger datasets. To achieve this, we plan to distribute the work evenly among the cluster, based on the estimates we calculate from previous iterations.

REFERENCES

- [1] Stephen D Bay and Michael J Pazzani. 2001. Detecting group differences: Mining contrast sets. *Data mining and knowledge discovery* 5, 3 (2001), 213–246.
- [2] Roberto J Bayardo Jr. 1998. Efficiently mining long patterns from databases. *ACM Sigmod Record* 27, 2 (1998), 85–93.
- [3] Guillaume Bosc, Chedy Raïssy, Jean-François Boulicaut, and Mehdi Kaytoue. 2016. Any-time diverse subgroup discovery with monte carlo tree search. *arXiv preprint arXiv:1609.08827* (2016).
- [4] Vladimir Dzyuba and Matthijs van Leeuwen. 2017. Learning what matters—Sampling interesting patterns. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, 534–546.
- [5] Vladimir Dzyuba, Matthijs van Leeuwen, and Luc De Raedt. 2017. Flexible constrained sampling with guarantees for pattern mining. *Data Mining and Knowledge Discovery* 31, 5 (2017), 1266–1293.
- [6] Henrik Grosskreutz and Stefan Rüping. 2009. On subgroup discovery in numerical domains. *Data mining and knowledge discovery* 19, 2 (2009), 210–226.
- [7] Robert J Hilderman and Terry Peckham. 2005. A statistically sound alternative approach to mining contrast sets. In *Proceedings of the 4th Australia Data Mining Conference (AusDM-05)*. 157–172.
- [8] Rohan Khade, Jessica Lin, and Nital Patel. 2015. Frequent Set Mining for Streaming Mixed and Large Data. In *Machine Learning and Applications (ICMLA), 2015 IEEE 14th International Conference on*. IEEE, 1130–1135.
- [9] Rohan Khade, Nital Patel, and Jessica Lin. 2015. Supervised Dynamic and Adaptive Discretization for Rule Mining. In *SDM Workshop on Big Data and Stream Analytics*.
- [10] Ming-Yen Lin, Pei-Yu Lee, and Sue-Chen Hsueh. 2012. Apriori-based frequent itemset mining algorithms on MapReduce. In *Proceedings of the 6th international conference on ubiquitous information management and communication*. ACM, 76.
- [11] Michael Mampaey, Siegfried Nijssen, Ad Feelders, and Arno Knobbe. 2012. Efficient algorithms for finding richer subgroup descriptions in numeric and nominal data. In *Data Mining (ICDM), 2012 IEEE 12th International Conference on*. IEEE, 499–508.
- [12] Gurmeet Singh Manku and Rajeev Motwani. 2002. Approximate frequency counts over data streams. In *Proceedings of the 28th international conference on Very Large Data Bases*. VLDB Endowment, 346–357.
- [13] Petra Kralj Novak, Nada Lavrač, and Geoffrey I Webb. 2009. Supervised descriptive rule discovery: A unifying survey of contrast set, emerging pattern and subgroup mining. *Journal of Machine Learning Research* 10, Feb (2009), 377–403.
- [14] Kostas Patroumpas and Timos Sellis. 2006. Window specification over data streams. In *International Conference on Extending Database Technology*. Springer, 445–464.
- [15] Matthijs van Leeuwen and Arno Knobbe. 2012. Diverse subgroup set discovery. *Data Mining and Knowledge Discovery* 25, 2 (2012), 208–242.
- [16] Geoffrey I Webb. 1995. OPUS: An efficient admissible algorithm for unordered search. *Journal of Artificial Intelligence Research* 3 (1995), 431–465.
- [17] Geoffrey I Webb, Shane Butler, and Douglas Newlands. 2003. On detecting differences between groups. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 256–265.
- [18] Geoffrey I Webb and Songmao Zhang. 2005. K-optimal rule discovery. *Data Mining and Knowledge Discovery* 10, 1 (2005), 39–79.
- [19] Jeffery Xu Yu, Zhihong Chong, Hongjun Lu, and Aoying Zhou. 2004. False positive or false negative: mining frequent itemsets from high speed transactional data streams. In *Proceedings of the Thirtieth international conference on Very large data bases—Volume 30*. VLDB Endowment, 204–215.

Recalibration of Analytics Workflows

Maxim Filatov

Yandex, Moscow, Russia
maxfil@yandex-team.ru

Verena Kantere

School of Electrical and Computer Engineering
National Technical University of Athens
verena@dbl.ece.ntua.gr

ABSTRACT

As business decisions and strategies become more and more automated, real-time, and data-driven, enterprises need to create, manage and execute end-to-end analytics workflows that process increasing data volumes, from new heterogeneous data sources, on specialized processing engines. Workflows become more complex and time-consuming to design and execute, since they span a variety of systems and the amount of data being processed grows. Therefore, it becomes increasingly difficult to debug workflows in order to handle errors, as well as adjust the workflow design and calibrate task parameters for applications that perform exploratory data analysis. Towards this end, the workflow management system should provide *recalibration* methods i.e. methods to monitor and to influence workflow processing at runtime. We demonstrate novel manual and automatic recalibration techniques for analytics workflows, on real use cases and data from the telecommunication domain and web analytics, but also on synthetic use cases and data.

1 INTRODUCTION

The analysis of Big Data is a core and critical task in multifarious domains of science and industry. Such analysis needs to be performed on a range of data stores, both traditional and modern, on data sources that are heterogeneous in their schemas and formats, and on a diversity of query engines. Workflow execution can be extremely resource- and time-consuming. Thus, a system that enables such long-term analytics processes on Big Data needs to be able to show the progress of the execution and the intermediate results. This means that the user should be able to monitor which workflow tasks have been executed, their produced results, which tasks are currently executing, as well as data accessing and resource utilization based on input from the runtime machines or the visualization tool. Further, the system should allow the user to influence workflow processing. This means that the system should provide methods that enable the analytics expert to change a workflow by altering task parameters or infusing new tasks manually at runtime, or even to predefine automatic changes while she creates the workflow by providing alternative workflow branches. Such *recalibration* methods constitute a powerful functionality of a workflow management system, since they enable the gradual design of exploratory analytics workflows based on feedback from intermediate results, as well as efficient error handling of complex and long-running workflows.

In this demonstration we focus on the novel functionality of PAW¹ (Platform for Analytics Workflows) for workflow recalibration. PAW is a platform for the design, management, analysis,

optimization and execution of analytics workflows. The first version of PAW is presented in [1, 2] and includes the functionalities of workflow design and analysis in order to clarify execution semantics, single workflow optimization and multi-workflow optimization. In this demonstration we present for the first time the new functionality of PAW on workflow recalibration. It includes novel techniques for (a) manually changing a workflow at runtime and re-executing it avoiding repeated computations, called *recovery and monitoring points* technique (3.1); (b) automatically changing a workflow at runtime based on conditional structures *if-then-else* (3.2) and *goto* statements (3.3).

Several existing workflow management systems support conditional structures to some extent. Each of these systems implements these structures in different ways and with some limitations: Kepler [7] allows the design of scientific workflows and executes them efficiently using emerging Grid-based approaches to distributed computation. It offers a structure called *Comparator*, which takes two inputs and performs a numerical comparison. Taverna [8] is a well-known workflow management system that does not include conditional structures in the workflow model, but tries to achieve the *if* and *switch* functionality at a higher layer of workflow management. In Taverna such conditional behavior is implemented using processors *fail_if_false* and *fail_if_true* placed as first vertices of parallel branches. Depending of their input one of those processors fails, another satisfies and only satisfied branch continue execution. UNICORE is a grid middleware, aims to provide seamless, secure and intuitive access to distributed resources [9]. UNICORE has a programming environment to design and execute workflows. It supports three specific types of *if-then-else* conditions, *ReturnCode*, *FileTest* and *TimeTest*. The first performs a numerical comparison, the second checks if a file exists or is executable and the third checks the current time. Recalibration in PAW offers an abstract *if-then-else* task that can be customized for a variety of input data and complex conditions that involve the execution of fully-fledged procedures. Only Taverna offers the same level of flexibility in the design of conditions as PAW does. The rest of the considered systems are very limited in possibilities to construct a condition. Also, they don't provide *goto* statements.

A new system that offers interactive debugging framework for big data processing is BigDebug [10], which provides a set of debugging primitives: (a) simulated breakpoints and on-demand watchpoints that allow users to selectively examine intermediate data of computation; (b) data provenance capability, crash culprit determination, tracing and (c) capability to fix code at runtime by the user, avoiding a program re-run from scratch. Unlike PAW, BigDebug is a single-engine system and works only on top of Apache Spark. Moreover, debugging is only one of the several uses of the manual recalibration of PAW.

Overview of PAW. PAW implements a novel workflow model [4, 5]. A workflow W is a directed, acyclic graph (DAG) $G = (V, E)$. The vertices V represent data processing tasks T and the edges E represent the flow of data. Each task is a set of *inputs*, *outputs* and an *operator*. Data and operators need to be accompanied by a set

¹Source code and live demo can be found on
<https://github.com/project-asap/workflow>

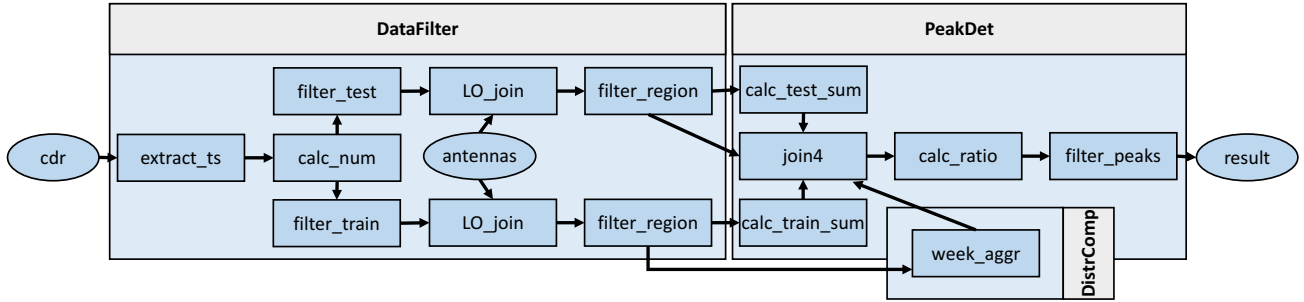


Figure 1: ‘Peak Detection’ of mobile calls workflow

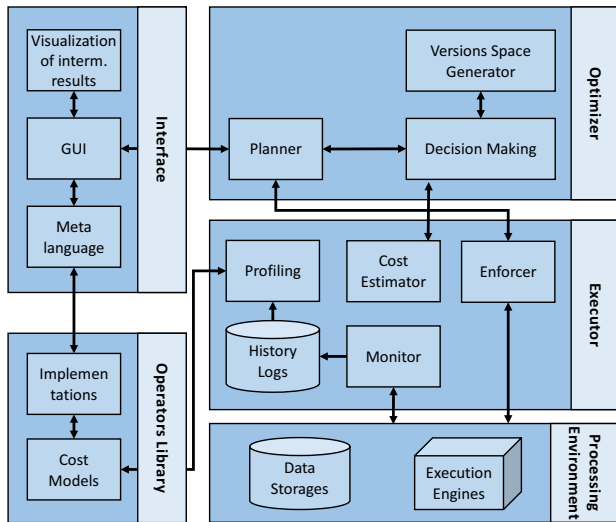


Figure 2: The architecture of PAW

of metadata, i.e., properties that describe them. Such properties include input data types and parameters of operators, the location of data objects or operator invocation scripts, data schemas, implementation details, engines etc. PAW is a part of the system ‘Adaptable Scalable Analytics Platform’ (ASAP) [3], but it can also stand as an independent tool for workflow management and optimization. Figure 2 depicts the architecture of PAW, as well as its interaction with the rest of ASAP. ASAP components include execution, visualization of results, online adaptation etc. PAW enables workflow design by users with various expertise, the automation of workflow analysis in order to clarify and specify execution semantics, single and multiple workflow optimization with respect to time efficiency, over a diverse collection of data stores and processing engines, monitoring of workflow execution and manual and automatic workflow recalibration.

2 MOTIVATING EXAMPLE

Figure 1 shows a real-world analytics workflow from a telecommunication company, which involves processing of the anonymised Call Data Records (CDR), collected in Rome for 2015 year and stored in HDFS, to populate a report on a dashboard. The report lists peaks in calls and their ratios to an averaged number of calls over a training period (one month). Peaks are defined by “differences from typical”. The workflow extracts the day of the week and the hour of the day from the timestamp of each call record (*extract_ts*). The task *calc_num* counts the number of calls at one-hour intervals. Then, two filters split the data to training

and test datasets. Further, the analysis is limited to specific geographical regions and, then, the number of calls in the training period is averaged over each mobile tower region, day in a week and hour in a day (*week_aggr*); this is the typical distribution of calls. Next, *calc_test_sum* and *calc_train_sum* produce the sum of calls in each day of the test and training datasets. Then, the test and training data are joined and the ratio of calls to the average number is produced. The *filter_peaks* finds ratios that are over a specific limit. These peaks is the sought information.

Initially, this workflow comprised three complex UDFs, (*DataFilter*, *DistrComp* and *PeakDet*), implemented in PySpark. Later on, for optimization needs [6], they were decomposed to smaller basic tasks, the operators of which have implementations in Spark and PostgreSQL. It is quite common that industrial workflows, like this one, are versioned and updated with time, resulting in a design that may not be optimal for the exploration procedure that the analytics expert needs to follow. In this example, the expert needs to explore the peaks one by one. This requires a complete restart of the workflow changing each time the search regions, a parameter of the *filter_region* tasks. This problem can be solved with recalibration methods that allow for the re-usage of the intermediate results produced by the tasks leading to the *filter_region* tasks, without re-executing the first. Also, recalibration methods would enable the expert to monitor the result of *filter_region*, visualized on a geographical map, so that she can observe faster call congestions and decide to change the search region. Furthermore, methods for automatic recalibration enable the expert to predefine conditions on intermediate results and, moreover, predefine decisions to be taken according to the outcome of condition evaluation.

3 WORKFLOW RECALIBRATION

We propose three techniques that perform recalibration in an online manner, i.e. during workflow processing.

Recovery and monitoring points. This technique offers to the user manual recalibration. It enables the user to monitor intermediate results, make workflow changes and if the changes are in the already executed workflow part, then re-execute only the changed part, avoiding to repeat computations; if changes affect only the non-executed part then workflow changes are applied and execution continues.

Conditional points. This automatic technique allows the execution of alternative predefined workflow branches.

Goto points. This automatic technique conditionally changes an executed workflow part to a predefined alternative and re-executes it.

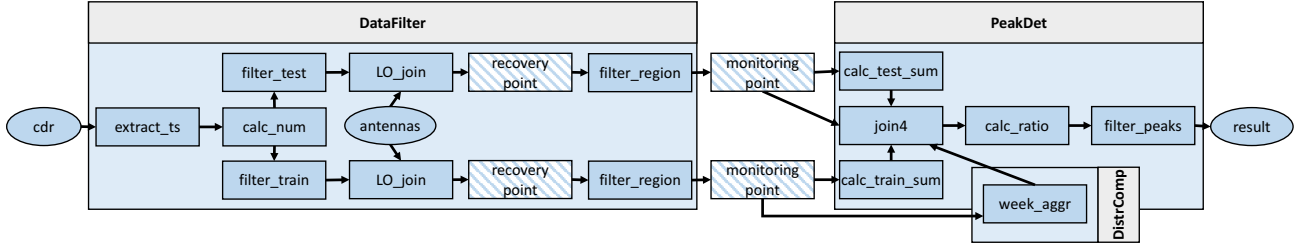


Figure 3: ‘Peak Detection’ workflow augmented with recovery and monitoring points



Figure 4: Monitoring intermediate results of ‘Peak Detection’ in PAW

3.1 Recovery and monitoring points

This recalibration technique allows the user to change a workflow during its execution and avoids to unnecessarily repeat computations in the already executed workflow part. It involves the employment of two novel types of tasks: recovery and monitoring points. A *recovery point* rp_T is a task that stores the result of task T . A *monitoring point* mp_T is a task that invokes the visualization of the result of task T or part of it. We use the phrase *intermediate result* to refer to the result of a task T that has been executed, while the whole workflow execution is not yet finished. The visualization of intermediate results assists the user in making a recalibration decision.

Recalibration using this technique is performed in four steps: (1) the user augments a workflow with recovery and monitoring points and starts the workflow execution; (2) when the execution reaches a recovery point the system stores the intermediate results of the preceding task, required for a possible re-execution of the workflow part following this recovery point; (3) when the execution reaches a monitoring point the user observes intermediate results of the preceding task; the workflow keeps executing after the monitoring point, while the user observes the intermediate results; (4) the user changes the workflow part following a recovery point and performs a re-execution of the workflow from this recovery point and on.

When the user changes the workflow and re-executes it, PAW determines which intermediate results are required to re-execute the changed workflow part that follows a specific recovery point (or points). It prepares this workflow part as a new materialized workflow with these intermediate results as input datasets and runs it. The execution of the previous (original) workflow is aborted.

Figure 3 displays the workflow from the motivating example augmented with recovery and monitoring points. The user observes the result of *filter_regions* at *monitoring points*, decides

to change the parameters of *filter_regions* and re-executes the workflow from the *recovery points*. So the most time-consuming part of *DataFilter* is not re-executed.

Figure 4 displays a ‘Peak Detection’ workflow in the interface of PAW. Green and yellow tasks are executed and currently executing, respectively. Using the geographical map on the bottom the user monitors intermediate results at the monitoring point marked with a blue stroke.

3.1.1 Implementations of points. Monitoring points invoke the visualization of the result of the preceding task. PAW includes monitoring points for specific operators, such as implementations of *k-means*, for which the result is visualized as a map of centroids or the histogram of cluster sizes. It also provides three basic monitoring operators, for the visualization of: geographical, numerical and categorical data. PAW includes recovery points for HDFS, Elasticsearch and PostgreSQL and operator-specific monitoring points for *k-means* and *tf-idf*.

3.1.2 Partial execution. We propose two improvements of the *recovery and monitoring points* technique:

Execution of a task until a deadline or a milestone is reached. The preceding task of a monitoring point is executed for a predefined time (deadline) or until it has processed a predefined amount of the input data (milestone). The partial intermediate result of the task is received by the following monitoring point, which invokes its visualization. This partial intermediate result is observed by the user, who decides on recalibration faster.

Execution on a part of a dataset. Data processing between recovery and monitoring points is made on a part of the input dataset. This enables the user to observe intermediate results on a part of the input data, and take the recalibration decision faster.

There is no general method for preparing operators so that they produce partial intermediate results. However, in some cases we can have a general methodology. For example, we can break up a query into multiple queries by dividing the input data in chunks, and then combine the results after the execution of all statements. PAW has several SQL query operators, which can be split up in this way.

```
SELECT * FROM db
WHERE field BETWEEN 1 AND 10000;
SELECT * FROM db
WHERE field BETWEEN 10000 AND 20000;
...
```

3.2 Conditional points

PAW includes a new type of task that realizes the conditional structure of the form *if-then-else*. The latter allows the design of a workflow with several alternative workflow parts. Depending on the intermediate results of the task preceding the *if-then-else* task,

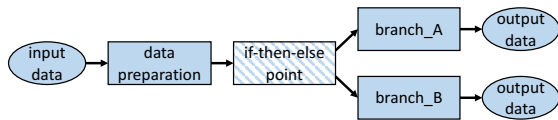


Figure 5: A workflow with an *if-then-else* point

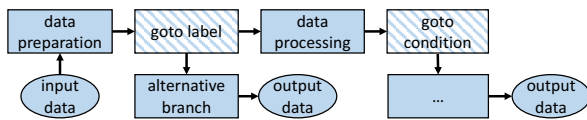


Figure 6: A workflow recalibrated with a *goto* point

a workflow branch is chosen for execution, over another one. These workflow branches are not yet executed. Figure 5 displays a workflow that has been augmented with one *if-then-else* point and two following workflow branches. The *if-then-else* task has two outputs; the boolean condition evaluates to true or false, depending on which PAW executes one of two branches.

The operator of the *if-then-else* point is implemented for any particular data. For example, for *tf-idf* PAW has an *if-then-else* task that evaluates if the weight of some word is above a certain value. Additional conditional points can be added through the interface of PAW.

3.3 Goto points

The workflow is augmented with two tasks: *goto label* and *goto condition* points, and an alternative workflow part related to the *goto label* point (Fig. 6). When the workflow execution reaches the *goto condition* point and if this task triggers ‘goto’ to *goto label*, then it re-executes the workflow from that point choosing for execution the alternative workflow part. Therefore, this technique is a combination of the *recovery and monitoring points* and *conditional* techniques.

The *goto condition* task has two outputs and a boolean condition evaluating to true or false, depending on which PAW continues execution or jumps to the *goto label* alternative workflow branch. The implementation of *goto condition* is similar to the *if-then-else* point.

4 DEMONSTRATION

In the following, we describe the proposed demonstration.

System setup. PAW is demonstrated on a cluster, with the following configuration: The cluster consists of 4 server-grade physical nodes. Each one of those is equipped with a 3rd generation i5 CPU (@ 2.90 GHz) and 16GB of physical memory and an array of two HDDs on RAID-0. The operating system is Debian 6 (squeeze) Linux. For the time being, four software platforms are running: Hadoop (CDH 4.6.0), Spark (1.4.1), Elasticsearch (5.1) and Weka (3.6.13).

Workloads. The demonstration uses synthetic and real workflows on real data. The real workflows and data come from the two use cases of ASAP [3] and belong to the domains of telecommunications and web analytics. One of the telecommunication workflows is described as a motivating example (Section 2). The web analytics use case involves anonymization of web content (WARC files) stored in Elasticsearch. The workflows are implemented in Spark and run over varying data set sizes ranging from 1 million to 4 billion rows. There are two types of workflows: one models entity recognition/disambiguation and k-means, and

another models continuous processing of incoming data, e.g., subscription/notification at scale.

Demonstration scenarios. The demonstration focuses on the recalibration functionality of PAW. It includes four types of scenarios that aim to show each a distinct view of the recalibration benefits and create discussion on the potential of recalibration of analytics workflows. The demonstration is interactive with the audience. The participants are invited to experience all functionalities of PAW, create workflows from scratch or change existing ones, watch the processing of the workflow, as well as review the internals of the platform, e.g. internal workflow representation. Even more, the participants are guided to play with the recalibration of workflows, by adding recovery and monitoring points, *goto* and *if-then-else* points, and change the workflow while monitoring intermediate results in GUI of PAW.

Scenarios A. These demonstrate the recovery and monitoring points technique. Specifically, they show real necessities to change workflows during execution. We show real workflows which need infusion of new tasks or alteration of task parameters during the execution.

Scenarios B. These also demonstrate the recovery and monitoring points technique. Specifically, they show how the user can design a workflow in a gradual and modular manner, while he is testing and debugging already created parts by monitoring intermediate results. We show how this workflow design process benefits exploratory data analysis.

Scenarios C. These demonstrate the *conditional* technique for workflows with a natural conditional branching, for which data analysis based on some conditions follows different paths, and the selection between these alternative paths should be made at runtime.

Scenarios D. These demonstrate the *goto* technique using workflows that benefit from the *goto* point in order to find anomalies in data, narrow or refocus the search or analysis, as well as meet deadlines and milestones of analysis.

REFERENCES

- [1] M. Filatov and V. Kantere. PAW: A Platform for Analytics Workflows. In *EDBT*, 2016.
- [2] M. Filatov and V. Kantere. Multi-workflow optimization in PAW. In *EDBT*, 2017.
- [3] Asap. <http://www.asap-fp7.eu/>.
- [4] V. Kantere and M. Filatov. A framework for big data analytics. In *C3S2E*, 2015.
- [5] V. Kantere and M. Filatov. Modelling processes of big data analytics. In *WISE*, 2015.
- [6] K. Doka, M. Filatov, V. Kantere and N. Papailiou. Optimizing, Planning and Executing Analytics Workflows over Multiple Engines. In *MEDAL*, 2016.
- [7] B. Ludascher, et al: Scientific Workflow Management and KEPLER System, Concurrency and Computation: Practice & Experience. SI on Scientific Workflows, 2005.
- [8] Taverna. <http://www.taverna.org.uk/>.
- [9] D.Erwin et al.: UNICORE Plus Final Report – Uniform Interface to Comp. Resources. The UNICORE Forum, 2003.
- [10] M. Gulzar, M. Interlandi, T. Condie and M. Kim: BigDebug: Interactive Debugger for Big Data Analytics in Apache Spark. in *FSE*, 2016.

Effective Quality Assurance for Data Labels through Crowdsourcing and Domain Expert Collaboration

Wei Lee

National Cheng Kung Univ.
Taiwan

wlee@netdb.csie.ncku.edu.tw

Chien-Wei Chang

National Cheng Kung Univ.
Taiwan

cwchang@netdb.csie.ncku.edu.tw

Po-An Yang

National Cheng Kung Univ.
Taiwan

payang@netdb.csie.ncku.edu.tw

Chi-Hsuan Huang

National Cheng Kung Univ.
Taiwan

chihsuan@netdb.csie.ncku.edu.tw

Ming-Kuang Daniel Wu

Slice Technologies Inc.
San Mateo, California, USA

danielwu@slice.com

Chu-Cheng Hsieh

Slice Technologies Inc.
San Mateo, California, USA

chucheng@ucla.edu

Kun-Ta Chuang

National Cheng Kung Univ.
Taiwan

ktchuang@mail.ncku.edu.tw

ABSTRACT

Researchers and scientists have been using crowdsourcing platforms to collect labeled training data in recent years. The process is cost-effective and scalable, but research has shown that the quality of truth inference is unstable due to worker bias, work variance, and task difficulty. In this demonstration, we present a hybrid system, named IDLE (Integrated Data Labeling Engine), that brings together a well-trained troop of domain experts and the multitudes of a crowdsourcing platform to collect high-quality training data for industry-level classification engines. We show how to acquire high quality labeled data through quality control strategies that dynamically and cost-effectively leverage the strengths of both domain experts and crowdsourcing.

1 INTRODUCTION

Hand-annotated training data, such as ImageNet ¹[2], have been the basis of many machine learning research. In recent years, crowdsourcing has become a common practice for generating training data [3], empowering researchers to outsource their tedious and labor-intensive labeling tasks to workers of crowdsourcing platforms. Crowdsourcing platforms provide large and inexpensive workforce for better cost control and scalability. However, the unstable quality of work produced by crowdsourcing platform workers is the main concern for crowdsourcing adopters.

Recent research by Zheng et al. [15] shows that the best truth inference algorithm is very domain-specific, and no single algorithm outperforms others in most scenarios. Sometimes an intuitive approach like an Expectation-Maximization algorithm could be a practical solution. In the literature, research advances focus on handling task difficulty [7, 13], worker bias [9], and worker variance [10, 12]. Specifically, *task difficulty* describes the degree of ambiguity of a question for which an annotated answer is sought; whereas *worker bias* and *variance* model the quality of

¹<http://www.image-net.org/>

workers – describing how likely a worker gives a wrong answer, assuming all tasks have equal difficulty.

The ability to collect high-quality and stable training data (i.e. the inferred truth) is essential for powering many supervised algorithms. These algorithms are often the foundation for modern business solutions, such as search engine rankings [6], image recognition [2, 11, 14], news categorization [8], and so on. Even though research [15] has unveiled the challenges of crowdsourcing labeling, it is undeniable that cost-effectiveness and scalability make crowdsourcing an attractive approach to generate training data.

In this study, we present a practical end-to-end multilevel solution based on a hybrid strategy. On the first level, we collect cost-effective truth inference from crowdsourcing workers whose answers have potentially high bias and variance.

On another level, we train a group of domain experts who are expected to perform labeling tasks with low worker bias and variance due to the training and financial incentives they receive. Our trained experts are intimately familiar with our product category taxonomy as well as the guidelines for assigning the most appropriate product category label to any given product item. They are instructed to mark high-difficulty tasks as “*unsolvable*” to circumvent ambiguous cases.

We propose IDLE as a system to facilitate the automated collaboration between our well-trained domain experts and the crowdsourcing workers to deliver high-quality hand-annotated training data. The IDLE framework streamlines the workflow for generating high quality training data by automating data filtration (by crowdsourcing) and data relabeling (by in house domain experts). It also provides an integrated environment for managing training data generation tasks as well as for assessing quality of classification results generated by our product classification engine as described in details in section 2.3.

2 SYSTEM FRAMEWORK

Figure 1 shows the architecture of our system. There are 4 key components in our framework: (1) **Multilevel Worker Platform**: a system that assigns tasks to domain experts and various crowdsourcing platforms through **Adapters**; it also performs **Worker Quality Assessment** and **Answer Aggregation**. (2) **Sampling Strategy**: with a unified user interface, job requester

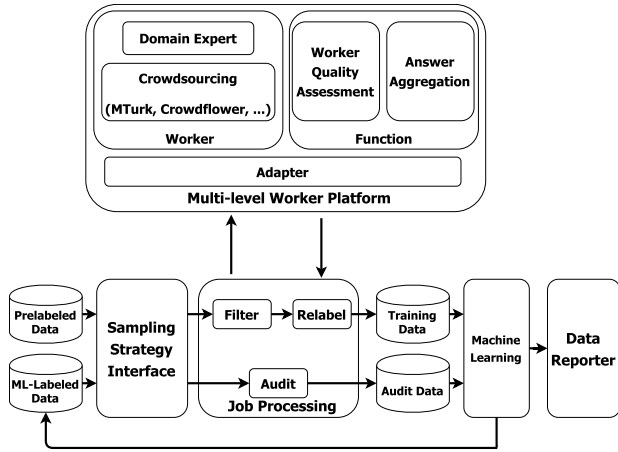


Figure 1: IDLE system framework

can choose among various sampling strategies. (3) **Job Processing**: job requester can launch jobs of various types. (4) **Data Reporter**: a dashboard showing the aggregated results from crowdsourcing and the improvement of the machine learning model.

2.1 Multilevel Worker Platform

With a unified interface, the job requester can submit a job through **Adapters** to various crowdsourcing platforms, such as MTurk² and Crowdfower³. Furthermore, the job requester can assign difficult labeling jobs to domain experts who sign into their **IDLE** account to label data. We also design a uniform **Function** interface for common features, such as **Worker Exclusion** and **Answer Aggregation**, across various crowdsourcing platforms.

Adapter: Using MTurk API⁴ as a reference, we design the interface through which job requester can (1) launch a job, (2) stop a job, and (3) retrieve results. Adapters allow us to easily integrate with different crowdsourcing platforms without making significant changes to the user experience or the rest of the IDLE platform.

Answer Aggregation: Since answers returned by crowdsourcing workers are not always consistent and worker’s quality varies (for example, master vs. non-master workers in MTurk), we have the challenge of inferring ground truth from the returned answers. To tackle the answer aggregation challenge, we implement three algorithms : **Majority Voting**[1], **Weighted Majority Voting**[4], and **Bayesian Voting**[5]. Through the provided interface, developers of IDLE platform can easily implement customized answer aggregation algorithms. Moreover, job requester can specify rules in the form of [#answer, #yes] for determining the final answer. The rule template is interpreted as seeking #yes/#answer level of consensus in total #answer number of answers. More elaborate answer aggregation strategies may be expressed through a sequence of rules. For instance, rules [3, 3] followed by [4, 3] together instruct the system to first seek unanimous consensus among 3 answers ([3, 3]). For questions whose answers fail to meet the first rule, the system needs to try again by soliciting an additional answer (#answer= 3 + 1) hoping to reach the specified 3/4 consensus.

²<https://www.mturk.com/>

³<https://www.crowdfower.com>

⁴<https://aws.amazon.com/documentation/mturk/>

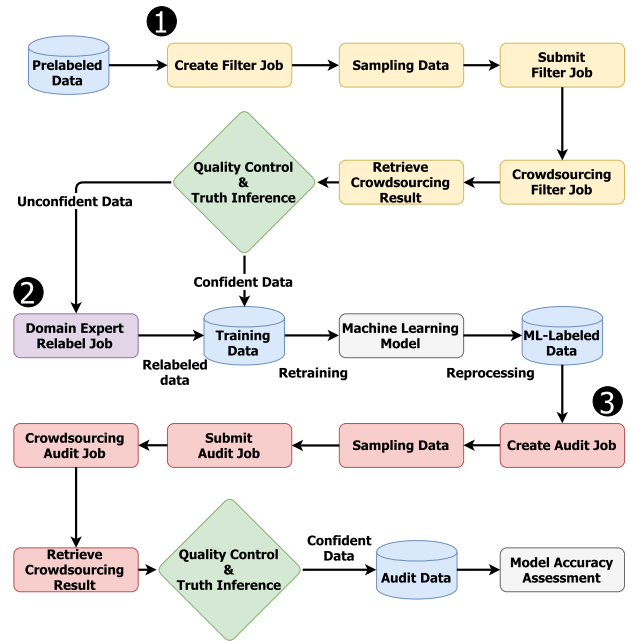


Figure 2: Control Flow

Worker Quality Assessment: Worker’s quality varies widely on crowdsourcing platforms. The fact that this quality is unknown to us in advance makes it even more important to assess worker’s quality. In IDLE, we randomly select questions from a curated pool of questions with ground truth answers (called *golden tasks*) to estimate worker’s quality. We apply two strategies: (1) **Qualification Test**: BEFORE performing the job, workers must first pass the *golden tasks*; (2) **Hidden Test**: mixing the *golden tasks* with the regular job questions, and we assess worker’s quality based on the *golden tasks* AFTER the job is completed. In our platform, job requester may use either one or both strategies to estimate worker’s quality.

2.2 Sampling Strategy Interface

There are many statistical sampling techniques. In IDLE, we design the general interface for developers to implement the required sampling strategies. The goal is for job requester to obtain sampling data from a diverse data set. We incorporate two hierarchical sampling strategies for IDLE in this version: (1) data clustering followed by stratified sampling; (2) topic modeling followed by stratified sampling.

2.3 Job Processing

As illustrated in Figure 2, there are three types of jobs in IDLE: Filter jobs, Relabel jobs, and Audit jobs.

Filter Job: A small set of data are sampled from pre-labeled data and sent to crowdsourcing platform for confirming their labels. Questions of a filter job are presented either as yes/no questions (e.g. Does the given label match this datum?) or multiple choice questions (e.g. Which of the following labels best matches this datum?). *Golden tasks* questions used for excluding poor-quality workers are also included in the filter job. After the workers submit their answers, the results are collected through the answer aggregation techniques described above in section 2.1. The results that are identified with high confidence level by our answer aggregation algorithm become new training data for the machine

learning model. The remaining (filtered-out) data are treated as mislabeled data and become input data for relabel jobs which are handled by domain experts as described in section 1. We expect data that are trivial for crowdsourcing workers can quickly pass through and data that are difficult to label are filtered out, hence, the name 'Filter' job. The cost of domain experts is much higher than that of crowdsourcing workers, which is why it is more cost-effective to have crowdsourcing workforce perform filter jobs on large number of trivial questions first and leave a small number of more challenging relabel jobs to domain experts.

Relabel Job: As mentioned above, mislabeled data are automatically collected and made available in IDLE framework to domain experts for relabeling. These domain experts are trained to assign correct labels to the provided data. Thus, data relabeled by domain experts do not require quality control or truth inference measures before they become training data for the machine learning model. With that said, there might be some data that even domain experts cannot label, thus are regarded as rejected data and recorded for further analysis.

Audit Job: After the filter job and the relabel job are done, all the sampled data are either identified as new training data for the machine learning model or as rejected data for analysis. After retraining the machine learning model in our product classification engine with the new training data, the model reprocesses data and updates the product category labels. Up to this point, all the efforts for enhancing the performance of the machine learning model are completed. We then assess the accuracy of this retrained model with an audit job. Similar to a filter job, a small set of data are sampled and sent to crowdsourcing platform for identifying correctly labeled data. We then apply our answer aggregation algorithm to identify data with high confidence level and calculate model accuracy while the mislabeled ones are simply discarded.

2.4 Data Reporter

To maximize the effectiveness of crowdsourcing and minimize the costs, there are certain questions that analysts would be curious about: for example, what is the ratio of filter job questions that need to be handled by a relabel job? The data reporter is a data visualization dashboard for administrators and analysts to evaluate the effectiveness of crowdsourcing and the performance of the machine learning algorithms. There are two parts of data reporter: (1) **Crowdsourcing Report** (2) **Machine Learning Model Report**.

Crowdsourcing Report: The purpose of crowdsourcing report is to evaluate the effectiveness and the efficiency of crowdsourcing. Therefore, it is designed to provide insights, such as the answer distribution and processing time. The crowdsourcing report includes the stats and results of crowdsourcing jobs. For filter jobs and audit jobs, the stats would include the ratio of YES vs. NO besides job completion time. For relabel jobs, the report would display the ratio of relabeled rate and job completion time. To estimate the overall performance of crowdsourcing for each job, the dashboard would also show the ratio of mislabeled data vs. data with high confidence level in addition to the total processing time.

Machine Learning report: The machine learning report is used to track the rate of improvement for the machine learning model. Thus, the report shows not only the history of accuracy for the model but also the ratio of data processed through crowdsourcing.

Algorithm 1 Adaptive Task Assignment

Require: Label confidence scores of prelabeled data $PC = (pc_1, pc_2, \dots, pc_n)$, task confidence threshold θ , worker set \mathcal{W} , budget B

Ensure: $(|W_1|, \dots, |W_n|)$

- 1: $(W_1, \dots, W_n) = (\emptyset, \emptyset, \dots, \emptyset)$
- 2: $PC_{order} = Order(PC)$ ▷ reverse sort PC
- 3: **for all** $pc_i \in PC_{order}$ **do**
- 4: $(W_i^*, C_i^*) = \underset{Conf(pc_i, W_i) \geq \theta, W_i \in \mathcal{W}}{\arg \min} Cost(W_i)$
- 5: $B = B - C_i^*$
- 6: **if** $B \leq 0$ **then**
- 7: **break;**
- 8: **return** $(|W_1|, \dots, |W_n|)$

3 ADAPTIVE TASK ASSIGNMENT (ATA) ALGORITHM

To best utilize available resources (such as a given budget), we further study the mechanism to adaptively assign the number of workers for each crowdsourcing task. In practice, the equal assignment of workers per task is not the most effective approach to achieve satisfactory quality when dispatching large-scale crowdsourcing tasks. Advances in machine learning research provide powerful labeling capabilities in predicting the label for each task along with a confidence score. Keeping the confidence granularity and the importance of a task in mind, we adjust the number of workers for each task in pursuit of overall optimization. Therefore, we propose **Adaptive Task Assignment Algorithm** to optimize the crowdsourcing resource utilization.

In Algorithm 1, we outline the steps to determine the number of workers for each task. Each crowdsourcing task consists of one single product item with category label predicted by our product classification engine. Initially, we are given prelabeled data and *label confidence score* pc_i of each task (provided by our supervised learning algorithms). *Label confidence score* ranges from 0 to 1. Next, we assign workers to crowdsourcing tasks that are reverse ordered by their label confidence scores. Here we introduce a threshold called θ , to ensure that the task confidence score of each task (calculated by the $Conf(pc, W)$ function described below) exceeds θ eventually. In addition, we use $Cost$ function to represent the cost for a worker set in search for the optimal worker set W_i^* , where the total cost C_i^* of the worker set is minimal for a task's confidence score to exceed θ . When the sum of cost C_i^* is equal to budget B , the algorithm terminates and returns the optimal number of workers $|W_i^*|$ to assign to each task.

The task confidence is calculated based on the given *label confidence* of prelabeled data and the quality of the assigned worker.

$$Conf(pc, W) = \max_{a \in \{Yes, No\}} Conf_a(pc, W)$$

We use label confidence score of prelabeled data pc_i and worker's quality q^w to calculate the Bayesian probability [5]. Worker's quality $q^w \in [0, 1]$ is the probability that the worker answers the correct label. As an example, assuming the supervised learning algorithm provides the prelabeled data with label confidence score of 0.45; the confidence threshold $\theta = 0.75$, and we have four workers with quality scores 0.5, 0.8, 0.6 and 0.4 respectively (assessed through golden tasks described above): Initially, we

pick the first two workers in the first iteration, and their answers are *No* (rejecting the label assigned by machine learning) and *Yes* (confirming the label assigned by machine learning). We can calculate the task confidence score as the following:

$$Conf_{Yes} \propto 0.45 \cdot (1 - 0.5) \cdot 0.8 = 0.18$$

$$Conf_{No} \propto (1 - 0.45) \cdot 0.5 \cdot (1 - 0.8) = 0.055,$$

which lead to

$$Conf_{Yes} = \frac{0.18}{0.18 + 0.055} = 0.76$$

$$Conf_{No} = \frac{0.055}{0.18 + 0.055} = 0.24$$

Since $Conf(pc_i, \{w_1, w_2\}) = 0.76$ exceeds $\theta = 0.75$, it is not necessary to assign additional workers to this task. In other words, the ATA algorithm can confidently confirm the machine-assigned label (*i.e.*, concluding the answer *Yes*) based on answers from merely two crowdsourcing workers.

4 DEMONSTRATION

Our system implementation of IDLE is based on Flask (a Python web framework) and ReactJS. We use distributed task queue Celery to handle asynchronous tasks, such as data sampling and crowdsourcing result retrieval. At the time of writing, IDLE is undergoing beta testing at Slice Technologies, and we are actively improving the system. The screenshots for the following demonstrations can be found in the GitHub repo⁵ of IDLE:

Data Ingestion: Job requester first selects pre-labeled data to ingest into IDLE by picking a category from a data set, uploading a file, or querying the database with SQL commands. Afterwards, job requester chooses a sampling strategy and sample count for filter job creation.

Crowdsourcing Job Configuration: After sampled pre-labeled data are ingested, job requester configures parameters of a crowdsourcing task, *e.g.* reward per assignment and number of assignments per HIT (Human Intelligence Task, a term to denote a single crowdsourcing task on Amazon Mechanical Turk platform). To estimate worker's quality, the system can also be configured to automatically include golden tasks (quality control questions) in the job.

Crowdsourcing Job Creation: Configurations of a crowdsourcing job are reviewed and confirmed prior to job creation. After publishing a job to crowdsourcing platform, IDLE automatically performs answer aggregation.

Stats Reporting: Job status and other job-related information are displayed on the main IDLE dashboard. History of the machine learning model's performance is also available for evaluation purposes.

5 CONCLUSIONS

In this study, we present IDLE, an integrated data labeling platform consisting of two main features: quality assessment and answer aggregation. The platform incorporates the adaptive task assignment algorithm, an algorithm that enables us to provide a cost-effective process for training data generation. This streamlined process alleviates the impact of highly difficult tasks as well as of crowdsourcing's worker bias and worker variance. As a result, IDLE system empowers researchers to effectively and efficiently collect high-quality training data through collaboration

between in-house domain experts and external crowdsourcing workers in an automated and integrated manner. IDLE provides us an integrated platform for generating large amount of training data with higher quality, faster speed, and optimal cost.

REFERENCES

- [1] Caleb Chen Cao, Jieying She, Yongxin Tong, and Lei Chen. 2012. Whom to Ask?: Jury Selection for Decision Making Tasks on Micro-blog Services. *Proc. VLDB Endow.* (2012), 1495–1506.
- [2] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on.* IEEE, 248–255.
- [3] Anhai Doan, Raghu Ramakrishnan, and Alon Y. Halevy. 2011. Crowdsourcing Systems on the World-Wide Web. *Commun. ACM* (2011), 86–96.
- [4] Chien-Ju Ho, Shahin Jabbari, and Jennifer Wortman Vaughan. 2013. Adaptive Task Assignment for Crowdsourced Classification. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28 (ICML '13)*. JMLR.org, 1–534–1–542.
- [5] Panagiotis G. Ipeirotis, Foster Provost, and Jing Wang. 2010. Quality Management on Amazon Mechanical Turk. In *Proceedings of the ACM SIGKDD Workshop on Human Computation*. ACM, 64–67.
- [6] Gabriella Kazai. 2011. In search of quality in crowdsourcing for search engine evaluation. In *European Conference on Information Retrieval*. Springer, 165–176.
- [7] Fenglong Ma, Yaliang Li, Qi Li, Minghui Qiu, Jing Gao, Shi Zhi, Lu Su, Bo Zhao, Heng Ji, and Jiawei Han. 2015. Faltcrowd: Fine grained truth discovery for crowdsourced data aggregation. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 745–754.
- [8] Richard MC McCreadie, Craig Macdonald, and Iadh Ounis. 2010. Crowdsourcing a news query classification dataset. In *Proceedings of the ACM SIGIR 2010 workshop on crowdsourcing for search evaluation (CSE 2010)*. 31–38.
- [9] Robin Wentao Ouyang, Lance Kaplan, Paul Martin, Alice Toniolo, Mani Srivastava, and Timothy J. Norman. 2015. Debiasing Crowdsourced Quantitative Characteristics in Local Businesses and Services. In *Proceedings of the 14th International Conference on Information Processing in Sensor Networks (IPSN '15)*. ACM, 190–201.
- [10] Vikas C Raykar, Shipeng Yu, Linda H Zhao, Gerardo Hermosillo Valadez, Charles Florin, Luca Bogoni, and Linda Moy. 2010. Learning from crowds. *The Journal of Machine Learning Research* (2010), 1297–1322.
- [11] Padhraic Smyth, Usama Fayyad, Michael Burl, Pietro Perona, and Pierre Baldi. 1994. Inferring Ground Truth from Subjective Labelling of Venus Images. In *Proceedings of the 7th International Conference on Neural Information Processing Systems (NIPS'94)*. MIT Press, 1085–1092.
- [12] Peter Welinder, Steve Branson, Serge Belongie, and Pietro Perona. 2010. The Multidimensional Wisdom of Crowds. In *NIPS (NIPS'10)*. Curran Associates Inc., 2424–2432.
- [13] Jacob Whitehill, Ting-fan Wu, Jacob Bergsma, Javier R Movellan, and Paul L Ruvolo. 2009. Whose vote should count more: Optimal integration of labels from labelers of unknown expertise. In *Advances in neural information processing systems*. 2035–2043.
- [14] Tingxin Yan, Vikas Kumar, and Deepak Ganesan. 2010. CrowdSearch: Exploiting Crowds for Accurate Real-time Image Search on Mobile Phones. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services (MobiSys '10)*. ACM, 77–90.
- [15] Yudian Zheng, Guoliang Li, Yuanbing Li, Caihua Shan, and Reynold Cheng. 2017. Truth Inference in Crowdsourcing: Is the Problem Solved? *Proc. VLDB Endow.* (2017), 541–552.

⁵<https://github.com/slice-ncku/IDLE>

Exploring Large Scholarly Networks with HERMES

Gabriel Campero Durand
University of Magdeburg, Germany
campero@ovgu.de

Anusha Janardhana
University of Magdeburg, Germany
anusha.janardhana@ovgu.de

Marcus Pinnecke
University of Magdeburg, Germany
pinnecke@ovgu.de

Yusra Shakeel
University of Magdeburg and
METOP GmbH, Germany
shakeel@ovgu.de

Jacob Krüger
Harz University and University of
Magdeburg, Germany
jkrueger@hs-harz.de

Thomas Leich
Harz University and METOP GmbH,
Germany
tleich@hs-harz.de

Gunter Saake
University of Magdeburg, Germany
saake@ovgu.de

ABSTRACT

Every year, the number of scientific publications increases, adding complexity to the networks of collaborations, citations, and topics, in which papers are embedded. Analyzing these networks with efficient tools is important to help researchers identify relevant works and understand scientific impact. However, available tools face several limitations, indicating that there is still room for improvement. We present HERMES, a prototype for exploring large and heterogeneous scholarly networks. HERMES allows users to seamlessly navigate diverse types of networks within a single graph, spanning hundreds of millions of nodes and relationships. Our prototype achieves reasonable responsiveness on commodity hardware through: *a*) comprehensive indexing, *b*) a careful coupling of a graph database and a search engine, and *c*) incremental processing of temporal queries. In this demonstration, we explain the techniques we adopt and illustrate how to use HERMES for exploring the MICROSOFT ACADEMIC GRAPH.

1 INTRODUCTION

The number of scientific publications increases every week, creating a large set of data about authors, citations, and collaborations. As a result, it becomes more and more challenging to determine which publications are relevant for a specific research goal [16]. In fact, new terms and fields, such as *big scholarly data* [22] and *science of science* [24], are emerging to cover data management challenges and novel analysis questions, which arise from scholarly information growth.

Scholarly network analysis (SNA): The traditional metrics of science (e.g., the *H-index*), that rely on network-unaware statistics, have been questioned by scholars [23], making a case for improving the analysis of scholarly networks by complementing such metrics with content & network-based analysis.

Types of scholarly networks: At least 7 types of networks are usually considered for SNA [23]; coauthorship/collaboration, citations, co-citations, bibliographical coupling, topics, co-words, and heterogeneous networks.

SNA and heterogeneous networks: Restrictive choices of network type and aggregation entity can limit the generalizability of SNA. To avoid this, researchers recommend to employ heterogeneous networks, and methods capable of extracting value from

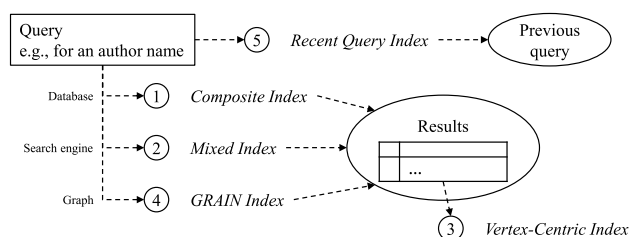


Figure 1: Indexing approaches in HERMES.

these networks [23]. One of such methods is FUTURERANK [15], an approach for relevance ranking based on combining HITS, over a collaboration graph, with PageRank, over a citation graph.

Data management for SNA: SNA involves studies at macro (global), meso (community), and micro (individual)-levels of a network. Considering the scale of networks, in addition to the need for managing heterogeneity of entities and analysis (e.g. content & statistical analysis), efficient tool support for SNA poses several data management challenges. Among them, the foremost could reasonably be resource management: to provision for ad-hoc SNA within reasonable response times, while enabling queries across different network scales and representations. Other challenges include data cleaning, provenance, management of analysis results, and integration with external sources.

Existing tools for large scale SNA: Several scientific digital libraries and search engines exist that index large amounts of publications [12], however their services often fall short in several aspects, including consistency, available metrics, and possibilities for ad-hoc SNA [11]. Regarding the latter, for example, ARNET-MINER¹ [20] internally utilizes several kinds of network analysis, however only local exploration of ego networks is currently offered to its Web users. Among tools supporting SNA, CITESPACE [4], PAJEK [2], GEPHI [1], IGRAPH, [6] and NETWORKX [10] are some of the most popular.

Graph databases and SNA: Although RDF technology has been widely researched for semantic publishing [3], to date there is little research in specialized graph database technologies for SNA. Within our work we consider this research gap.

In this demo paper we describe the first version of HERMES, our proposed tool for SNA based on graph technologies. The core technical insight behind our work at this stage is in exploring opportunities for close search engine/graph database coupling in SNA tasks.

¹<https://aminer.org/>

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

With our tool we seek to enable users to seamlessly interact with large scale heterogeneous networks, performing ad-hoc SNA at different granularities, either through our APIs, or through native graph/search-engine query languages.

Overall, we hope to encourage the audience to use our tool for multiple purposes. We aim to: 1) help the audience in identifying relevant publications and performing SNA tasks; 2) receive feedback to improve HERMES; 3) introduce to the audience the key indexing solutions that improve the performance of HERMES; and 4) employ HERMES for user studies to investigate what could constitute representative SNA workloads. Based on our work, we intend to publish corresponding anonymous datasets of usage patterns as open-source artifacts—thus providing data for workload characterization to the research community. Furthermore, our tool will be freely available.

2 HERMES

In this section we describe the current architecture of our tool, and we present two techniques we currently explore in HERMES for improving its data access.

2.1 Architecture

For indexing and primary storage, HERMES relies on a search engine, i.e., ELASTICSEARCH, and a database, i.e., CASSANDRA. These two components are integrated through a property graph database, JANUSGRAPH²—which can also be configured to other back-end indexes and storages. JANUSGRAPH uses the TINKERPOP³ framework, and the corresponding GREMLIN query language. Data access in HERMES is provided via a web-interface with data management supported by GREMLIN and ELASTICSEARCH servers.

JANUSGRAPH provides three predefined types of indexes:

- *Composites* are traditional indexes tuned for point-queries. They rely on the primary storage. They cover one or multiple keys of either vertexes or edges of the stored graph.
- *Mixed* indexes are based on the search engine. They enable inexact querying, and relevance boosting.
- *Vertex-centric* indexes are supported by the primary storage. They are included in the same storage space of a vertex. Instead of indexing all entries in the database (global)—as the former two indexes—they only index the set of edges attached to a vertex (local). This allows to traverse faster through edges while filtering on labels and properties.

Apart from these predefined indexes, users of property graph databases can also use the "graph itself as an index". This is a straight-forward approach, which we, for lack of a better term, call GRAIN (graph as an index) [8]. In GRAIN meta-vertexes are added to the modeled data, acting as roots and nodes of a search tree, such that they lead within limited hops to a set of expected entities. A practical enhancement is to denormalize properties of the target vertexes unto the linking edges, such that filtering can be applied on the edges without visiting any vertex. In our system, the GRAIN approach is additionally adopted by using a set of meta-vertexes with temporal expiration. They serve as a cache for previous query results.

Within HERMES, we adopt a comprehensive indexing strategy that exploits all these opportunities. In Figure 1, we show an overview. We remark that the numbering corresponds only to the order in which we introduced each index.

²<http://janusgraph.org/>

³<http://tinkerpop.apache.org/>

Listing 1: Alternative temporal queries with GREMLIN.

```
titanGraph.V().
  hasLabel("journal").
  has("retiredAt", P.gte(year0)).
  has("createdAt", P.lte(year0)).count().next();

titanGraph.V().
  hasLabel("journal").
  has("createdAt", year0+1).count().next();
```

2.2 Incremental Processing for Temporal Queries

Analyzing the evolution of publication networks can yield interesting findings. For example, studies on the MICROSOFT ACADEMIC GRAPH [18] have found the number of publications per year to be following an exponential growth for the last century—doubling every 12 years, with the top 1% of publications continuously accounting for around a quarter of citations each year [7]. Similar analyses with more localized perspectives can have practical value for researchers. For instance, they could help to assess the impact of given conferences over time or to identify high impact research topics.

Several works consider the formalization and evaluation of temporal graph queries—both, in graph databases [17], and in processing engines [13, 14]. Building on studies in this area [9], we design our temporal queries in HERMES by creating a type of meta-vertex, which we call *logger vertexes*. The set of edges in such vertexes provide a relative time-line for when items appear in the world represented by a graph. Analyzing the sequence of edges in a logger vertex allows reconstructing a graph from a certain point in time to another. Through this, it is possible to implement incremental computation of temporal data over a graph model that accumulates several snapshots.

We achieve incremental computation in HERMES by hand-tuned query rewriting. In Listing 1, we provide an example of a GREMLIN query for identifying the number of journals on a year-by-year basis throughout a specified period. Instead of adding up the number of items for all years within the interval, we only compute the existing journals for the first year. Then, we keep a rolling total, based on the number of journals created or retired in each successive year.

Recently, authors have considered automated rewriting of multi-snapshot queries for co-scheduling tasks by their common steps—in a style similar to SIMD processing [21]. The authors call this approach *Single Algorithm Multiple Snapshots*. Unlike their work, we do not store separate snapshots, and we do not include automated rewrites in our tool.

In other study, groundwork for *incremental view maintenance* on property graph databases has been proposed, over a formalization of OpenCypher [19]. In contrast, the rewrites for incremental processing illustrated with our current prototype are hand-tuned, and require further standardization.

2.3 Query Rewrites Across Graph and Search Engine Representations

Similar to other graph databases, such as NEO4J⁴, JANUSGRAPH can integrate a search engine to support full-text queries. This provides a set of search engine functionalities, which are either

⁴<https://neo4j.com/developer/elastic-search/>

Listing 2: Rewrite of degree centrality (DC) calculation for ELASTICSEARCH.

```
TermsBuilder termsBuilder = AggregationBuilders.terms(aggregationName).
    field("idOfTargetVertex").size(numberOfVertices);

XContentBuilder contentBuilder;
contentBuilder = JsonXContent.contentBuilder().startObject();
termsBuilder.toXContent(contentBuilder, ToXContent.EMPTY_PARAMS);
contentBuilder.endObject();

SearchRequestBuilder searchRequest= esClient.prepareSearch("index").
    setTypes("edges").
    setQuery(QueryBuilders.termQuery("edgeLabel", "label")).
    setAggregations(contentBuilder);
response = searchRequest.execute().actionGet();
```

offered through the graph API (e.g., launching so-called *index-Queries*), or directly through the search engine APIs. After verifying the potential of *external* access in previous work [8], we seek to leverage this concept in HERMES to improve its functionality by rewriting queries for the search engine APIs.

To make the case for such rewriting, we consider an example degree centrality (DC) calculation in GREMLIN that has to be rewritten for the search engine. The DC of vertexes is a measure to understand the actors of a network. It can be defined as the number of edges (either incoming or outgoing) connected to vertexes. The in-DC of a given vertex indicates how *prominent* it is within the network, while the out-DC indicates how *influential* it is. The average vertex DC for a graph is the average of the DCs of its vertexes. In GREMLIN, we can express the in-DC computation as a traversal that starts from all vertexes with an incoming edge having a specific label—grouping them by the vertex ID, and the number of edges. Here is an implementation based on recommendations by the TINKERPOP community⁵:

```
g.V().in("label").inV().group().by(____.ID()).by(____.inE(label).count())
```

For the search engine, we can increase the utility of the indexed data by adding to each edge the JANUSGRAPH ID of the connected vertexes. This design decision amounts to a limited use of denormalization, with few chances for inconsistency as the IDs of connected vertexes do not change. Finally, we can construct the average DC calculation as a single ELASTICSEARCH term query that searches for a given value over all edge labels. This query aggregates the count of results based on the IDs of vertexes (either the source or the target vertex). The result is an ordered list of vertexes and number of edges to which they are either the source or the target. We show a prototypical implementation in Listing 2.

For an initial evaluation of this rewrite we use the Pokec dataset, an online social network, and a commodity machine⁶. In Figure 2, we show the results, which indicate a speedup of 150x over the original, by employing the second query approach.⁷

The fundamental reason for the differences in performance is that the queries in fact map to entirely different algorithms. A GREMLIN traversal begins by querying a given set of vertexes (either using a composite index or a full-table scan). For each match, GREMLIN counts the number of incoming edges with a specific label, groups the results by vertex IDs, and returns these

⁵<http://tinkerpop.apache.org/docs/3.2.1-SNAPSHOT/recipes/#degree-centrality>

⁶We used an Intel Core™ i7-2760QM CPU (2.40GHz) processor with 8 cores and 7.7 GiB of memory.

⁷We would like to add the disclaimer that the performance gains we report are specific to the queries and database we selected. Further comparisons are pertinent to assess the benefits of these rewrites for other queries and databases.

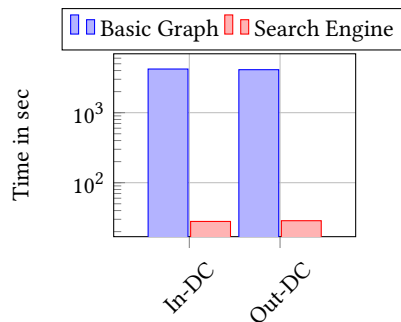


Figure 2: Response times for computing the average degree centrality (DC) on the Pokec dataset.

values. In contrast, the ELASTICSEARCH implementation relies on a simpler representation of the same data: A set of documents that stand for the edges, further indexed with LUCENE’s inverted indexes. These serve well term-based searches, such as our *entry query*. Another index, covering as terms, the ID of source vertexes, can be matched with the document numbers found by our *entry query*. At last, the resulting collection is aggregated by counting for each term the number of matching edge documents.

In spite of the benefits, there are limits that need to be considered in adopting a search engine for graph tasks in a multi-store context. Namely, the *impedance mismatch* between domains, *inadequate performance optimizations* in the search engine, and *overheads from application-level mapping* of items between the different storage engines.

3 DEMONSTRATION OVERVIEW

Data: Our preloaded dataset is the complete MICROSOFT ACADEMIC GRAPH [18] as of early 2017, which is a property graph of scholarly publications. It is compiled and made publicly available by *Microsoft*. The graph consists of six entities: Field of study, author, institution, paper, venue, and event. Furthermore, the graph includes six standard relationships: Citations among papers, paper authorship, the venue or journal a paper was published at (or a field of study if no venue is available), the institutional affiliations of authors, and the relationships of events to venues. Overall, the MICROSOFT ACADEMIC GRAPH comprises around 166 million papers, representing a heterogeneous and large dataset for our tool.

Considering that the audience of our demonstration will be conformed by the database community, we have tagged some subgraphs of interest within the dataset. This includes a graph of papers with transitive reference relationships to the foundational paper of Edgar Codd on relational databases [5]. We call this the *Codd’s world* graph, on which we aim to undertake studies in future work.

Demonstration: We will start introducing the audience to the core functionalities of HERMES by searching for authors or papers, as shown in Figure 3. Based on the results, we navigate through some different network representations of the data and explore dependencies between the results. This will enable the audience to use the tool by themselves.

In the next step, we explain some representative scenarios for scholarly network analysis: *Impact evaluation*, *academic recommendation*, or *expert identification*. This selection is based on a recent survey [22]. We then carry out a task from the outlined cases. For every alternative, we first accomplish the task and then walk the user through the series of queries that conform it.

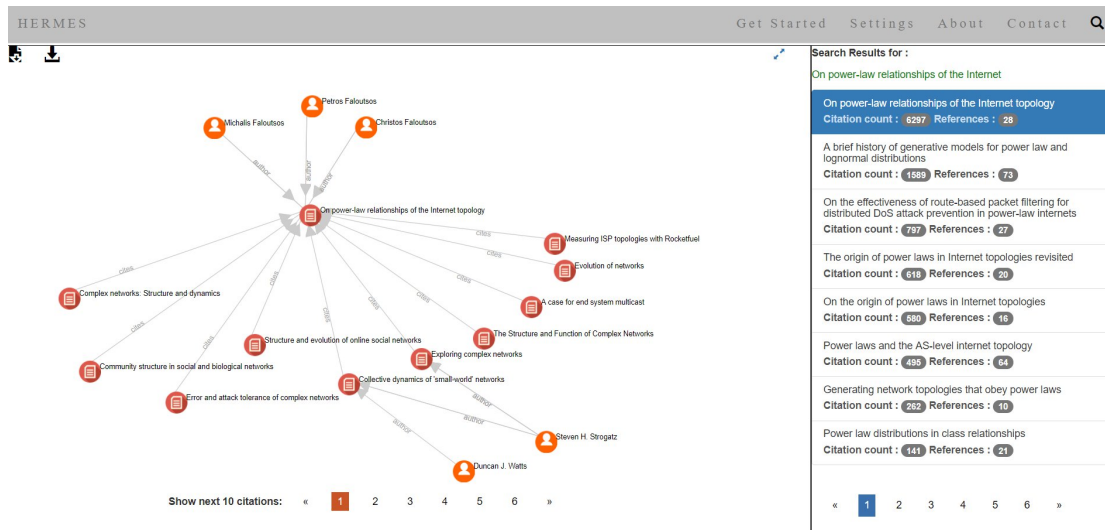


Figure 3: Exploring networks with HERMES.

Following this, we take the user to a JUPYTER notebook, where we submit alternative versions (with different optimizations) of a single query. For the queries we observe different execution times in spite of them achieving the same task. This provides insights into the practical, positive impact of the optimizations.

As a takeaway, we expect the user to understand the different indexing and incremental processing techniques, which can be used to build efficient prototypes based on mainstream graph databases. We hope to encourage them to use our tool when searching for literature, due to its supportive features.

4 CONCLUSION

In this paper we introduce HERMES, a tool for exploring and analyzing large scholarly datasets building upon on a graph database and an integrated search engine. While in an early phase, it already comprises a range of functionalities that motivate us to share our work with the community. We seek to encourage others to use our tool, helping us to improve it and, through user studies, contributing to the understanding and characterization of SNA workloads. For future work, we will enhance HERMES with additional analysis options and automated query rewrites. Other target features include support for cross-dataset exploration and for connecting to more data sources (e.g., ORCID). Potentially, we can also extend our tool to the purpose of supporting systematic literature reviews and research exploration, addressing the lack of suitable tools in these research areas [11, 16].

ACKNOWLEDGMENTS

We would like to thank the reviewers for their insightful feedback. We thank Jingy Ma for helping us to efficiently load large graph data. This work is supported by DFG grants SA 465/50-1 and LE 3382/2-1, and the DAAD STIBET Matching Funds grant.

REFERENCES

- [1] Mathieu Bastian, Sebastien Heymann, Mathieu Jacomy, et al. 2009. Gephi: an open source software for exploring and manipulating networks. *Icwsm* 8 (2009), 361–362.
- [2] Vladimir Batagelj and Andrej Mrvar. 1998. Pajek-program for large network analysis. *Connections* 21, 2 (1998), 47–57.
- [3] Peter Boncz. 2013. LDBC: Benchmarks for Graph and RDF Data Management. In *IDEAS*. ACM, 1–2.
- [4] Chaomei Chen. 2006. CiteSpace II: Detecting and Visualizing Emerging Trends and Transient Patterns in Scientific Literature. *Journal of the Association for Information Science and Technology* 57, 3 (2006), 359–377.
- [5] Edgar F. Codd. 1970. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM* 13, 6 (1970), 377–387.
- [6] Gabor Csardi and Tamas Nepusz. 2006. The igraph software package for complex network research. *InterJournal, Complex Systems* 1695, 5 (2006), 1–9.
- [7] Yuxiao Dong, Hao Ma, Zhihong Shen, and Kuansan Wang. 2017. A Century of Science: Globalization of Scientific Collaborations, Citations, and Innovations. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 1437–1446.
- [8] Gabriel Campero Durand. 2017. *Best Practices for Developing Graph Database Applications: A Case Study Using Apache Titan*. Master’s thesis. University of Magdeburg.
- [9] Gabriel Campero Durand, Marcus Pinnecke, David Broneske, and Gunter Saake. 2017. Backlogs and Interval Timestamps: Building Blocks for Supporting Temporal Queries in Graph Databases. In *EDBT/ICDT Workshops*.
- [10] Aric Hagberg, Pieter Swart, and Daniel S Chult. 2008. *Exploring network structure, dynamics, and function using NetworkX*. Technical Report. Los Alamos National Laboratory (LANL).
- [11] Edgar Hassler, Jeffrey C. Carver, David Hale, and Ahmed Al-Zubidy. 2016. Identification of SLR Tool Needs - Results of a Community Workshop. *Information and Software Technology* 70 (2016), 122–129.
- [12] Narayanan Meyyappan, Gobinda G. Chowdhury, and Schubert Foo. 2000. A Review of the Status of 20 Digital Libraries. *Journal of Information Science* 26, 5 (2000), 337–355.
- [13] Youshan Miao, Wentao Han, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Enhong Chen, and Wenguang Chen. 2015. Immortalgraph: A System for Storage and Analysis of Temporal Graphs. *ACM Transactions on Storage* 11, 3 (2015), 14.
- [14] Vera Zaychik Moffitt and Julia Stoyanovich. 2016. Towards a Distributed Infrastructure for Evolving Graph Analytics. In *WWW. WWW Steering Committee*, 843–848.
- [15] Hassan Sayyadi and Lise Getoor. 2009. Futurerank: Ranking Scientific Articles by Predicting Their Future Pagerank. In *SIAM ICDM*. SIAM, 533–544.
- [16] Ivonne Schröter, Jacob Krüger, Philipp Ludwig, Marcus Thiel, Andreas Nürnberg, and Thomas Leich. 2017. Identifying Innovative Documents: Quo Vadis?. In *ICEIS*. ScitePress, 653–658.
- [17] Konstantinos Semertzidis and Evaggelia Pitoura. 2017. Historical Traversals in Native Graph Databases. In *ADBIS*. Springer, 167–181.
- [18] Arnab Sinha, Zhihong Shen, Yang Song, Hao Ma, Darrin Eide, Bo-june Paul Hsu, and Kuansan Wang. 2015. An Overview of Microsoft Academic Service (MAS) and Applications. In *WWW*. ACM, 243–246.
- [19] Gábor Szárnyas. 2017. Incremental View Maintenance for Property Graph Queries. *arXiv preprint arXiv:1712.04108* (2017).
- [20] Jie Tang, Jing Zhang, Limin Yao, Juanzi Li, Li Zhang, and Zhong Su. 2008. ArnetMiner: Extraction and Mining of Academic Social Networks. In *KDD*. ACM, 990–998.
- [21] Manuel Then, Timo Kersten, Stephan Günemann, Alfons Kemper, and Thomas Neumann. 2017. Automatic Algorithm Transformation for Efficient Multi-Snapshot Analytics on Temporal Graphs. In *Proceedings of the VLDB Endowment*, Vol. 10. VLDB Endowment, 877–888.
- [22] Feng Xia, Wei Wang, Teshome Megersa Bekele, and Huan Liu. 2017. Big Scholarly Data: A Survey. *IEEE Transactions on Big Data* 3, 1 (2017), 18–35.
- [23] Erjia Yan and Ying Ding. 2014. Scholarly Networks Analysis. In *Encyclopedia of Social Network Analysis and Mining*. Springer, 1643–1651.
- [24] An Zeng, Zhesi Shen, Jianlin Zhou, Jinshan Wu, Ying Fan, Yougui Wang, and H Eugene Stanley. 2017. The Science of Science: From the Perspective of Complex Systems. *Physics Reports* (2017).

Don't mix pages with different lifetimes in one stream

Soyee Choi
Sungkyunkwan University
Suwon, Kyounggi, Republic of Korea
ithdli@skku.edu

Hyun-Woo Park
Sungkyunkwan University
Suwon, Kyounggi, Republic of Korea
music2eye@skku.edu

Sang-Won Lee
Sungkyunkwan University
Suwon, Kyounggi, Republic of Korea
swlee@skku.edu

ABSTRACT

We present a demonstration about optimizing two database storage engines by leveraging multi-streamed SSDs (MS-SSD in short). By storing data pages with similar lifetime together in the same physical flash blocks, MS-SSD can effectively reduce the overhead of garbage collection, improving the write performance and prolonging the lifespan. Thus, in order to benefit from MS-SSD, it is very crucial for database storage engines to precisely classify logical data pages according to their update intervals and to effectively map those logical data streams to physical streams in MS-SSD. Given that numerous new interfaces between host and flash memory SSD for better performance are emerging, this demonstration will provide a model case of physical database tuning on flash memory SSDs.

We have successfully multi-streamed two database engines, MySQL/InnoDB and ForestDB, by identifying several logical data streams with distinct update intervals in each engine and by taking the *stream-per-object* policy, instead of the naïve *stream-per-file* one. By running both vanilla and multi-streamed versions of two storage engines on real MS-SSD, we showcase that multi-streamed versions consistently outperform vanilla ones. In addition, we propose a set of guidelines on how to group logical streams with different update intervals into smaller number of physical streams with minimal performance degradation.

1 INTRODUCTION

During the last decade, we are witnessing that flash memory SSDs have relentlessly been replacing harddisks as the main storage because of several advantages such as high IOPS/\$ and low power consumption [9]. However, to prevent data loss due to electrical interference, flash memory chips do not allow overwrite. Hence, a costly erase operation against a block is necessary prior to overwriting the existing data pages in the block [5]. For this reason, most contemporary flash storage device takes the log-structured copy-on-write approach and, among many FTL schemes, the page-mapping FTL approach is most popular [10]. In FTLs, when no more clean block is available, a costly but inevitable garbage collection (GC) operation has to be triggered so as to secure new blocks to write new incoming page writes. During GC, valid pages from the victim block has to be copybacked to a clean block. It is well known that the excessive copybacks of valid pages during GC negatively affects the performance and lifetime of flash memory SSDs. Hence, one of the key challenges in modern flash memory SSDs is to reduce the GC overhead.

Meanwhile, every modern flash memory SSD has abundant computing resource which is affordable to support other new interfaces than the existing dummy read and write block interface. In fact, numerous new interfaces between host and flash

memory SSD have been actively proposed for various purposes including better performance. Among them, one notable interface is Multi-Streamed SSD (MS-SSD in short) [8]. The interface is recently standardized in the SCSI interface [11] and the commercial SSDs which support it exists. The goal of the MS-SSD is to minimize the GC overhead. That is, by storing data pages with different lifetimes in different physical flash memory blocks (i.e., different write streams), MS-SSD expects that it can reduce, compared to the existing non-multi-streamed SSD, the number of pages to be copybacked in victim blocks, and thus can improve both the write performance and the lifespan. In short, instead of writing all data pages in one stream regardless of lifetime, by explicitly allowing to cluster data pages with different lifetime into different write streams, the MS-SSD interface is intended to reduce write amplification due to GC.

However, the performance benefit of MS-SSD depends heavily on the accuracy of classifying logical data streams according to update interval [7]. Therefore, it is very crucial for storage engines to precisely classify logical data pages base on their update intervals and to effectively map those logical data streams to physical streams in MS-SSD. For this reason, the first step in making any database engine multi-streamed is to understand its write patterns and then to figure out all logical data streams distinguishable from each other in terms of update intervals.

In this demonstration, we will show how to make two database storage engines multi-streamed, MySQL/InnoDB and ForestDB, and present the benefit of each multi-streamed version over its vanilla engine in terms of transaction throughput and WAF. The contributions of this demonstration can be summarized as follows. First, we show that each database storage engine has several logical data streams with distinct update intervals. Second and more importantly, we show that, in database engines, the logical data streams with different update intervals can be found when the write patterns are analyzed *per-object*. Unlike the existing work on multi-streaming LSM-based NoSQL engines such as Cassandra and RocksDB using the *per-file* policy [6, 8], we found out that those two database engines used in this demonstration can not be effectively multi-streamed with the *per-file* policy. Given that numerous interfaces between host and flash memory SSDs for better performance are emerging, this demonstration will provide a model case of physical database tuning on flash memory SSDs. Our demonstration will proceed following the steps below:

- Using the logical data streams identified according to the *per-object* approach in each storage engine, we explain how to classify each of streams into *Hot*, *Cold* and *Warm* and the rationale behind it. (Section 2)
- Based on the classification obtained from the above step, we will show that there are numerous other combinations in mapping logical streams into physical streams than the naïve one-to-one mapping, run representative benchmark in each storage engine by changing the combinations, explain the results, and discuss its implications. (Section 4.2)

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

- While running benchmarks on both multi-streamed and vanilla version of each storage engine, we will show, using a GUI program, how the key metrics including CPU utilization, IOPS, TPS (transaction per second) and WAF dynamically change over time. (Section 4)
- Based on the performance results from several combinations in mapping logical streams to physical streams, we suggest a set of practical guidelines for making storage engines multi-streamed effectively. (Section 4.2)

2 BACKGROUND

2.1 MultiStream SSD

The goal of MS-SSD is to reduce the GC overhead by separating logical data pages with different lifetimes into different physical *streams* of flash blocks inside SSD [8]. Multiple *physical streams* will divide physical space in flash SSDs into several smaller spaces. Applications in the host are responsible for distinguishing data pages by explicitly attaching *stream-id* when making a write request to MS-SSD. Upon receiving write request for data page(s) with *stream-id*, SSD places the page(s) in the flash block belonging to the corresponding physical stream id. All the blocks belonging to each physical stream will be managed by the flash translation layer (FTL) separately from other blocks of other physical streams. Consequently, compared to the non-multi-streamed SSDs, MS-SSD expects that most pages in victim blocks upon GCs is invalidated for GC, thus minimizing the number of pages to be copybacked for GCs.

Although MS-SSD looks promising, there are at least two practical issues to be addressed when making any database engine multi-streamed. As noted above, the *stream-id* of data pages is not determined automatically by MS-SSD itself, but instead should be explicitly hinted by applications. Thus, the performance benefit of any multi-streamed database engine will be highly dependent on the accuracy of logical data stream classification. Next, because the number of physical streams available in an MS-SSD is limited in practice (e.g., 16 in the case of PM953), applications should be able to get best performance with the limited number of physical streams. For this, when the number of logical data streams from the applications is larger than that of physical streams supported by MS-SSD, a set of guidelines on how to group multiple logical streams into smaller number of physical streams with minimal performance degradation.

2.2 Logical Streams in Database Engines

As discussed above, the crux in leveraging the opportunities from MS-SSD is to accurately separate logical stream with different lifetime. In this section, we illustrate how logical streams from each of two real database engines, MySQL/InnoDB and ForestDB, are derived. From a set of separate experiments, where two storage engines were, likewise as in existing work [8] multi-streamed according to the *per-file* approach, any meaningful performance improvement was not observed. In some cases, the performance of multi-streamed versions was even worse than that of non-streamed vanilla ones. This is because the write patterns from those database engines do not reveal any distinguishable lifetime among different database files.

MySQL/InnoDB is a popular open source relational database engine, which takes the traditional in-place update policy. On the other hand, ForestDB, a storage engine for Couchbase NoSQL database [3, 4], is taking the out-of-place update

approach, likewise other popular NoSQL engines such as Cassandra and RocksDB. But unlike these LSM-based NoSQL engines used in the previous work on MS-SSD [8], ForestDB is a B-tree-based storage engine, which appends new key-value versions at the end of files (that is, copy-on-write). It periodically reuses the space occupied by the invalidated old versions of key-value documents and, when the size of a database file becomes larger, the compaction operation has to be carried out. In this section, although these two database engines of MySQL/InnoDB and ForestDB take different approaches in updating data, they are common in that each engine has several object types and in turn each object type exhibits distinct update intervals. This observation clearly confirms that there exist opportunities for improving database performance by making those engines multi-streamed using the *stream-per-object* approach.

Table 1: Characteristics of MySQL TPC-C’s Data Type

	avg update interval	total write (MB)	write ratio(%)
DWB (H1)	2	1330556	50
new_orders(H2)	226410	88349	3.32
order_line(C1)	26425310	202777	7.62
customer(C2)	7741410	157006	5.54
orders (W)	2861170	108706	4.085
stock (W)	2873060	771190	28.98

2.2.1 Mysql. In order to derive logical data streams, which are suitable to the purpose of MS-SSD, from MySQL/InnoDB engine we collected the write trace while running TPC-C benchmark [1]) with 200GB database size for one day. Using the trace, we calculated the average update interval, total write amount, and the relative write ratio for major object types in the database. Between the time point when data is written and updated in each LBA, write commands are issued to another LBAs. Average update interval represents the average of the numbers of intervening writes issued for all LBAs belonging to each object type. Therefore, the larger average update interval is, the less frequently the pages in the object type is updated. The results for major object types (with write ratio greater than 1%) are summarized in Table 1.

The most outstanding object from the table is *double-write buffer* (DWB), to which every dirty page evicted from the buffer cache has to be redundantly journaled to guarantee the page write atomicity. It shows very low value of average update interval and also occupies half of total writes in MySQL/InnoDB. For this reason, DWB is definitely a *hot* data object with very short lifetime and is thus denoted as H1 in Table 1. In addition, we see from the table that the *new_orders* table has relatively low average update interval and thus it is also regarded as *hot object*. Similarly, two tables, *order_line* and *customers* are treated as *cold objects* and all other object are as *warm objects*.

2.2.2 ForestDB. A ForestDB database consists of multiple files and each database file is comprised of four data types: database header, super block, index node, and document. As mentioned above, there was no performance gain when multi-streamed using the *stream-per-file* approach. Therefore, as in the case of MySQL/InnoDB, in order to verify that those four object

Table 2: Characteristics of ForestDB’s Data Type

	avg update interval	total write (MB)	write ratio(%)
DB Header (W1)	764571	6069	10.4
Index Node (W2)	848531	1048	1.8
Data Page (C)	1457589	44858	77.1
Super Block (H)	752	6222	10.7

types are suitable as logical data streams for MS-SSD, we collected the write trace while running ForestDB-Benchmark workload [2] with 7.5GB database for four hours. Using the trace, we calculated the average update interval, total write amount, and the relative write ratio for those four object types. The results are given in Table 2. From Table 2, it is obvious that Super Block is hot (denoted as H), DB Header and Index Node warm (denoted as W1 and W2, respectively), and Data Page cold (denoted as C. In addition, we verified that all data pages of each object type tend to have uniform update intensity.

3 SYSTEM OVERVIEW

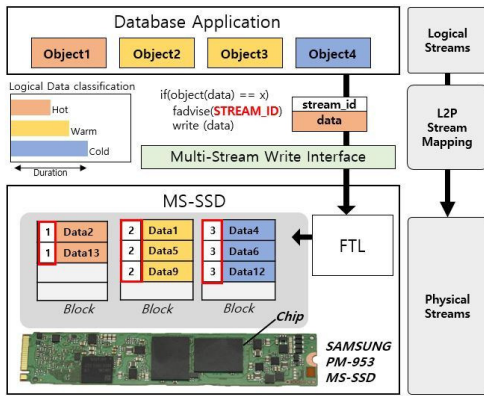

Figure 1: Multi-Streamed Database Engine: Architecture

Figure 1 shows the architectural overview of how a multi-streamed database engine interacts with MS-SSD. First, as shown in the bottom of Figure 1, a commercial MS-SSD from Samsung with NVMe interface (PM953 model) is used. The MS-SSD currently provides 16 *physical streams*. Next, as shown at the top of Figure 1, a multi-streamed database engine will, for each data page to be written, identify the *logical data stream* where the page belongs according to the *stream-per-object* policy explained in Section 2. Finally, as illustrated in the middle of Figure 1, the database engine is responsible for mapping its each logical stream to a specific physical stream in MS-SSD using the `posix_fadvise` system call. For example, the database engine can assign different physical stream to its each logical stream (that is, one-to-one mapping between logical and physical stream). As another example, as exemplified in Figure 1, four logical data streams in database engine can be combined to map to three different physical streams in MS-SSD (that is, many-to-one mapping between logical and physical stream).

The database engine will, before writing a data page, first check its logical stream, then assign the appropriate `stream_id` to the page according to the mapping between logical and physical streams, and finally call

the multi-streamed write interface using the `ioctl` command of `posix_fadvise(fd, stream_id, 0, POSIX_FADV_STREAMID)`. Upon receiving the command, FTL will place the data page in the physical stream corresponding to the given `stream_id`.

4 DEMONSTRATION DETAIL

4.1 Demonstration Scenario

The main goals of this demonstration are two-folds. First, we will show that real database engine can significantly benefit by accurately classifying its logical streams according to the *stream-per-object* policy and then by calling the multi-stream interface. Second, given the limited number of physical streams available in real MS-SSDs, we will show that it is possible to achieve nearly optimal performance by effectively using physical streams less than logical streams.

Table 3: Stream Combinations (MySQL/InnoDB)

5 streams	2 streams	
(H1, H2, C1, C2, W)	(H1, else)	(C1, else)
	(H1+H2, else)	(H1+C1, else)

Table 4: Stream Combinations (ForestDB)

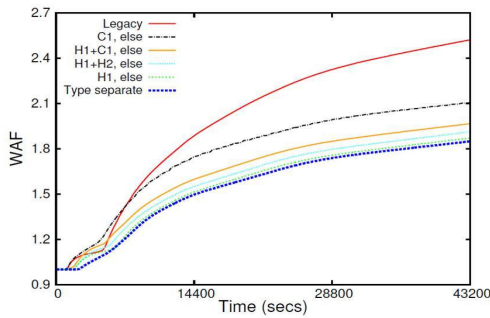
4 streams	3 streams	2 streams	
H, C, W1, W2	W1+W2, H, C	(H, else)	(C, else)
	H+C, W1, W2	(W1, else)	(W2, else)

For this, by running TPC-C an ForestDB-Benchmark on MySQL/InnoDB and ForestDB, respectively, this demonstration will present the performances of vanilla version of each storage engine. In addition, we will present, as the baseline performance, the performance of its multi-streamed version when run by assigning one physical stream to each logical stream. As shown in the first column of Table 3 and Table 4, respectively, a dedicated physical stream is assigned to each logical stream in Table 1 and in Table 2, respectively. Then, for each database engine, we will present the performance of multi-streamed version when run by grouping logical data streams into smaller number of physical streams in several meaningful combinations. In the case of MySQL/InnoDB, we tested all four combinations shown in the second column of Table 3. For example, the combination (H1+H2, else) in the table represents that two hot logical streams of H1 and H2 share one physical stream while all other three logical streams do other physical stream. Similarly, in the case of ForestDB, we tested all the six combinations shown in the second and third columns of Table 4.

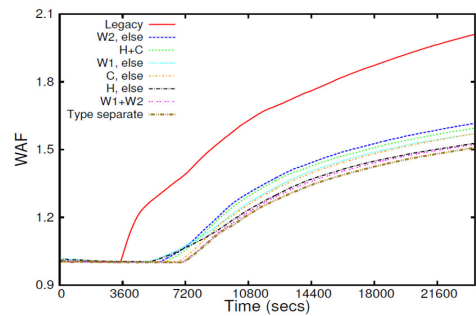
In order to show the effect of multi-streamed database visually, we made a GUI system to monitor status of computer resources utilization, which is illustrated in Figure 3. Using the GUI we will compare the effect of multistream SSD.

4.2 Preliminary Performance Evaluation

For each of MySQL/InnoDB and ForestDB engines, we have evaluated the performance of its multi-streamed version as well as its vanilla version. In the case of MySQL/InnoDB, we measured the write amplification factors over time while running



(a) TPC-C on MySQL/InnoDB



(b) ForestDB-Benchmark on ForestDB

Figure 2: Preliminary Experimental Results: Original vs. Multi-Streamed Database Engine

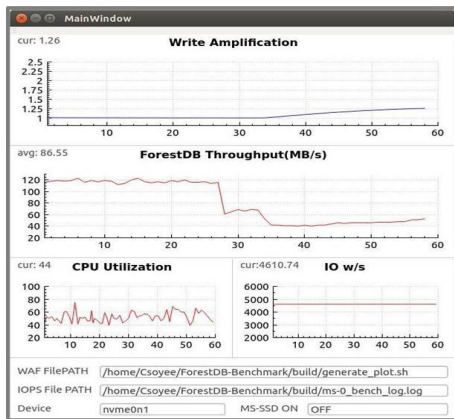


Figure 3: GUI used in Demonstration

TPC-C benchmark for twelve hours on its multi-streamed version for every five combinations in Table 1 as well as on its non-multi-streamed version. The results are presented in Figure 2(a). Similarly, in the case of ForestDB, we measured the write amplification factors over time while running ForestDB-Benchmark for six hours on its multi-streamed version for every seven combinations in Table 2 as well as the original ForestDB version. The results are presented in Figure 2(b).

From Figure 2(a) and Figure 2(b), we can make several common observations on the performance implications of combining logical streams into physical streams. First, since every multi-streamed versions always outperforms the vanilla version for both database engines, it is, obviously, always beneficial to separate at least one logical stream. Second, the best performance is achievable by one-to-one mapping between logical and physical streams. Third, it is better to combine data objects having similar lifetime rather than different lifetime. For example, when comparing the (H1+H2, else) case with the (H1+C1, else) one in the case of MySQL/InnoDB, the former case shows lower WAF value. Also, the (W1+W2, else) combination in ForestDB outperforms all other combinations except for one-to-one mapping logical and physical mapping case, in terms of WAF value. Fourth, it is always desirable to separate logical streams with extremely low update interval, such as DWB (H1) in MySQL/InnoDB and SuperBlock(H) in ForestDB. Lastly, though obvious, it is less effective to separate any logical stream with very low write ratio than to separate one with high write ratio, as exemplified by two streams of W1 and W2 in ForestDB.

5 CONCLUSION AND FUTURE WORK

In this demonstration, we have shown that database engines can significantly benefit from MS-SSD by appropriately identifying logical streams according to the *stream-per-object* policy. In addition, given that the number of physical streams available in real MS-SSDs is limited, we have derived a set of guidelines on effectively grouping logical streams into the fewest physical streams with minimal performance degradation.

One promising future research direction is to automatically identify logical streams out of any write-intensive application, which are suitable to MS-SSD, considering that we found out a set of logical streams from each of two database engines manually. Another challenging future work is to automatically group logical streams into minimum number of physical streams.

ACKNOWLEDGEMENTS

This research was supported in part by IITP under the “SW Starlab” (IITP-2015-0-00314) and in part by Samsung Electronics.

REFERENCES

- [1] 2008. tpcc-mysql benchmark. <https://github.com/Percona-Lab/tpcc-mysql>. (2008).
- [2] 2014. Forest Database System Benchmark. <https://github.com/couchbaselabs/ForestDB-Benchmark>. (2014).
- [3] 2014. ForestDB. <https://github.com/couchbase/forestdb>. (2014).
- [4] Jung-Sang Ahn, Chiyong Seo, Ravi Mayuram, Rahim Yaseen, J.W Kim, and Seungryoul Maeng. 2016. ForestDB: A Fast Key-Value Storage System for Variable-Length String Keys. *IEEE Trans. Comput.* 65 (March 2016), 902–915.
- [5] Shingo Nishioka Atsuo Kawaguchi and Hiroshi Motoda. 1995. A Flash-Memory Based File System. In *UniSex Winter*. 155–164.
- [6] Yang Fei, Dou Kun, Chen Siyu, Hou Mengwei, Kang Jeong-Uk, and Cho Sangyeon. 2015. Optimizing NoSQL DB on Flash: A Case Study of RocksDB. In *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing*.
- [7] Yang Jingpei, Pandurangan Rajinikanth, Choi Changho, and Balakrishna Vijay. 2017. AutoStream: automatic stream management for multi-streamed SSDs. (May 2017).
- [8] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoon Maeng, and Sangyeun Cho. 2014. The Multi-streamed Solid-State Drive. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*.
- [9] Sang-Won Lee, Bongki Moon, and Chanik Park. 2009. Advances in Flash Memory SSD Technology for Enterprise Database Applications. In *Proceedings of the 35th SIGMOD international conference on Management of data*. 863–870.
- [10] Ma, Dongzhe and Feng, Jianhua and Li, Guoliang. 2014. A Survey of Address Translation Technologies for Flash Memories. *ACM Computing Survey* 46, 3, Article 36 (Jan. 2014), 36:1–36:39 pages.
- [11] William Martin (T10 Technical Editor). 2015. SCSI Block Commands - 4 (SBC-4) (Working Draft Revision 9): 4.34 Stream Control. (November 2015), 110–112 pages. <http://www.t10.org/cgi-bin/ac.pl?t=f&f=sbc4r09.pdf>

ELINDA: Explorer for Linked Data

Tal Yahav Oren Kalinsky Oren Mishali Benny Kimelfeld

Technion – Israel Institute of Technology
Haifa 32000, Israel

ABSTRACT

To realize the premise of the Semantic Web towards knowledgeable machines, one might often integrate an application with emerging RDF graphs. Nevertheless, capturing the content of a rich and open RDF graph by existing tools requires both time and expertise. We demonstrate ELINDA—an explorer for Linked Data. The challenge addressed by ELINDA is that of understanding the rich content of a given RDF graph. The core functionality is an exploration path, where each step produces a bar chart (histogram) that visualizes the distribution of classes in a set of nodes (URIs). In turn, each bar represents a set of nodes that can be further expanded through the bar chart in the path. We allow three types of explorations: subclass distribution, property distribution, and object distribution for a property of choice. To efficiently compute the exploration queries, we offer a query engine powered by a worst-case-optimal join algorithm.

1 INTRODUCTION

The potential of enhancing Artificial Intelligence with rich human knowledge intensifies with the growth of Linked Data resources with high quality, volume, and wealth of domains. To realize this potential, developers continuously explore emerging datasets and investigate their relevance to the application at hand. We present ELINDA—an exploration tool for RDF that implements a novel visual query language for exploratory search, designed especially to facilitate comprehension of unfamiliar datasets. To that end, we exploit the ontology that is typically associated to an RDF graph, describing its semantics in terms of *classes*, *class hierarchies* and *properties*.

The formal model underlying our visual query language applies in an iterative manner the basic principle for effective data visualization by Shneiderman [6]: “Overview first, zoom and filter, then details-on-demand.” Specifically, our formal model is based on histograms over focus sets of nodes (URIs) that are constructed iteratively by the user. In this model a *bar chart* consists of a set of *bars*, each representing a portion of the focus set. The user selects a bar and applies an *expansion operation* that transforms a bar into a new bar chart that now focuses on the portion of the selected bar. In addition, a *filter* can be applied to restrict the bar chart according to a search condition. The user can then continue the exploration of the new bar chart, and hence, construct focus sets of arbitrary depths. (See Section 2 for the formal model.)

We illustrate the mission and functionality of ELINDA through a hypothetical exploration scenario over the DBpedia dataset [1]. Suppose that the user is interested in understanding what information DBpedia has on cities where scientists were born. The initial bar chart shows how *all* DBpedia nodes are distributed among the 49 top-level classes (see Figure 1). For example, the user can observe that the most popular classes are Agent and Work. The

user then selects Agent and applies a *subclass expansion* to get a histogram over agents. Two additional subclass expansions are then applied to focus on the Scientist nodes (through class Person). Next, a *property expansion* is applied to get the distribution of properties of scientists, and from that the user selects the birthPlace bar. An *object expansion* over this bar results in the histogram over the birth places of scientists, and from there the user selects the City bar.

The basic implementation of ELINDA translates each expansion into a SPARQL query that is sent to an endpoint. A major challenge is the *execution cost*, since the queries often involve large numbers of nodes to extract and apply aggregates to, and a naïve translation to a black-box SPARQL engine yields impractical responsiveness. For example, computing the distribution of properties over all DBpedia nodes takes around 10.5 minutes on a standard Virtuoso endpoint. Therefore, we have implemented a novel query engine that is specialized (and restricted) to support ELINDA’s exploration model. For that, we adopt and extend the *Cached Trie Join* (CTJ) of Kalinsky et al. [5], an algorithm from the breed of *worst-case-optimal joins*, to an evaluation algorithm that we refer to as CTJ*. Specifically, CTJ* supports reachability queries and grouped aggregations. Our engine often achieves 1.5-2 orders of magnitude speedups compared to Virtuoso.

In addition to CTJ*, ELINDA supports a *remote* execution mode that works on Virtuoso endpoints that are accessible only through a standard Web interface. In that case, ELINDA accelerates responsiveness by retrieving data *incrementally* and *caching* results. ELINDA has an accompanying demonstration video that is available at <https://tinyurl.com/y9hu2gxl>.

Related Work. ELINDA is inspired mostly by LD-VOWL [8] that extracts ontological information from the actual RDF graph by sending SPARQL queries to the RDF endpoint. The results are visualized in a graph-based fashion. The fundamental difference between LD-VOWL and ELINDA is the iterative exploration model of the latter—we use the visual tool for iteratively constructing focus sets of arbitrary description depth (as described earlier). Another important difference is our handling of efficiency that does not have any correspondence in LD-VOWL. Many of the existing ontology visualization tools, such as FlexViz [3] and GLOW [4] visualize an ontology, yet independently of the data. On the other extreme, a family of tools known as *linked-data browsers* [2] are able to provide informative insights into the details of a dataset by supporting the exploration of individual nodes via properties and relations with other nodes. Examples include *Marbles*¹ and *Sig.ma* [7]. In contrast to ELINDA, browsers are appropriate in cases where the task at hand is to look for specific information from a dataset that the user is familiar with.

2 FRAMEWORK

In this section, we give the formal definition of the data and interaction model underlying ELINDA.

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

¹<http://mes.github.io/marbles/>

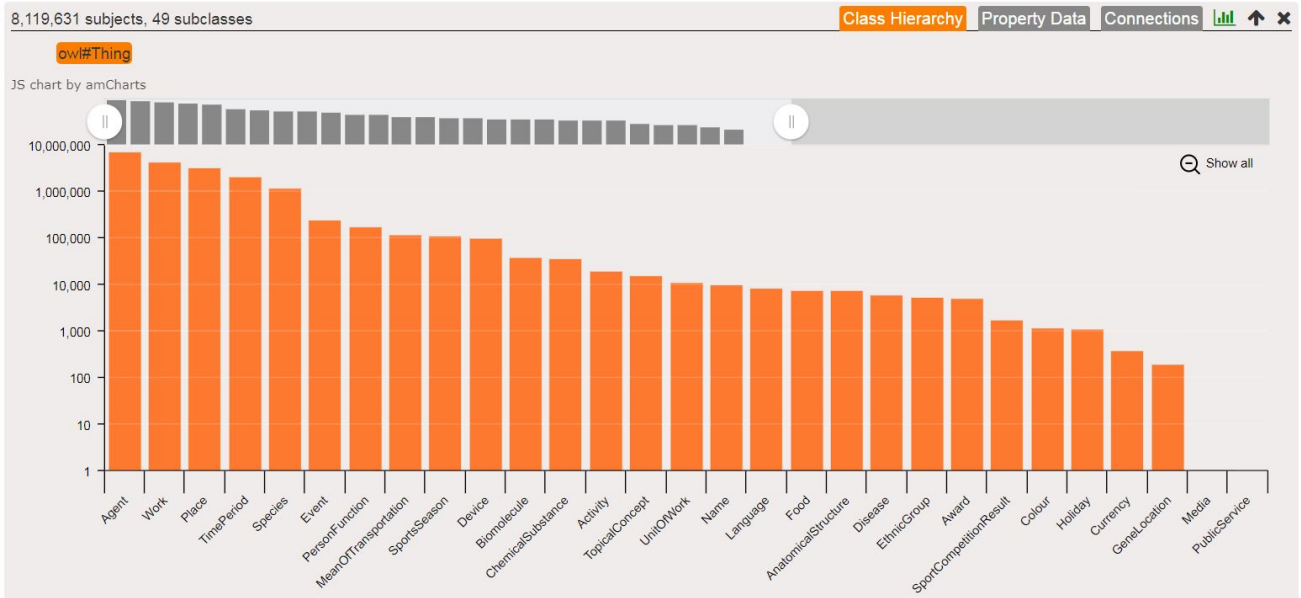


Figure 1: Initial chart in the exploration pane over DBpedia.

RDF graphs. We adopt a standard model of RDF data. Specifically, we assume collections U of *Unique Resource Identifiers* (URIs) and L of *literals*. An *RDF triple*, is an element of $U \times U \times (U \cup L)$. An *RDF graph* is a finite collection G of RDF triples. In the remainder of this section, we assume a fixed RDF graph G . A URI u is said to be *of class* c if G contains the triple $(u, \text{rdf:type}, c)$.

Bar charts. eLINDA enables the visual exploration of the RDF graph by means of bar charts that are constructed interactively (see Figure 2). We have two kinds of bars: a *class bar* represents URIs of a common class, and a *property bar* represents URIs with a common property. For a bar B , we denote by $U(B)$ the set of URIs represented by B . The *category* of B is the corresponding class or property, depending on the kind of B . A *bar chart* (or just *chart* for short) is a mapping from categories to bars.

Bar expansion. A *bar expansion* is a function E that transforms a given bar B into a chart $E(B)$. eLINDA supports three specific bar expansions E that we define as follows.

Subclass expansion: This expansion is enabled for class bars B ; in this case, the category c of B is a class. The categories of the chart $E(B)$ are all the *subclasses* of c , that is, the URIs c' such that G contains the triple $(c', \text{rdfs:subClassOf}, c)$. The bar $B_{c'}$ that $E(B)$ maps to category c' is a class bar with the category c' , and $U(B_{c'})$ consists of all the URIs $u \in U(B)$ such that u is of type c' .

Property expansion: This expansion is again enabled for class bars B . The categories of the chart $E(B)$ are the properties of the URIs of B , that is, the URIs p such that G contains (s, p, o) for some $s \in U(B)$. The bar B_p that $E(B)$ maps to p is a property bar with the category p , and $U(B_p)$ consists of all URIs $s \in U(B)$ that have the property p , that is, $(s, p, o) \in G$ for some o .

Object expansion: This expansion is enabled for property bars B ; in this case the category of B is a property p . The categories of the chart $E(B)$ are the classes c of the objects that are connected to the URIs in $U(B)$ through the property p ; that is, the classes c such that for some triple $(s, p, o) \in G$ it is the case that $s \in U(B)$ and o is of class c . The bar B_c that $E(B)$ maps to category c is a class bar with the category c , and $U(B_c)$ consists of the p -targets

of type c , that is, all URIs o of class c such that $(s, p, o) \in G$ for some $s \in U(B)$.

The property and object expansions are defined above for *outgoing* properties, that is, the URIs of B play the roles of the *subjects*. We similarly define the *incoming* versions, where the URIs of B play the roles of the *objects*.

Exploration. Finally, eLINDA enables the exploration of G by enabling the user to construct a list of charts in sequence, each exploring a bar of the previous chart. The exploration begins with a predefined *initial chart* that we denote by B_0 . In our implementation this bar has the form $E(B)$ where E is the subclass expansion and B is a bar that consists of all URIs of a predefined class. A sensible choice of such class is a general type such as `owl:Thing`. By *exploration* we formally refer to a sequence of the form $(c_1, E_1) \mapsto B_1, (c_2, E_2) \mapsto B_2, \dots, (c_m, E_m) \mapsto B_m$ where each chart B_i is obtained by selecting the bar B of category c_i from the chart B_{i-1} and applying to B the expansion E_i . As a feature, eLINDA enables the user to generate SPARQL code to extract each of the bars along the exploration.

In addition to the expansion tasks, eLINDA allows, at any stage, to *filter* the current chart by a filtering condition (e.g., the name of the node contains a certain string). The semantics is straightforward—every bar is restricted to the nodes that satisfy the condition.

3 USER INTERFACE

eLINDA is implemented as a single-page Web application that points to an online SPARQL endpoint hosting the explored data. During an exploration, eLINDA fetches data from the endpoint by sending numerous SPARQL queries. The user experience is *visual*, and no SPARQL knowledge is needed. The user should have only a basic understanding of ontology classes and properties.

eLINDA's basic UI component is a *tabbed pane* as in Figure 1. Each tab in the pane presents a specific bar chart, which is the result of an expansion applied on a bar of a previous pane. The opened tab in Figure 1 shows the initial subclass expansion for DBpedia. The bar chart visualizes the distribution of all DBpedia

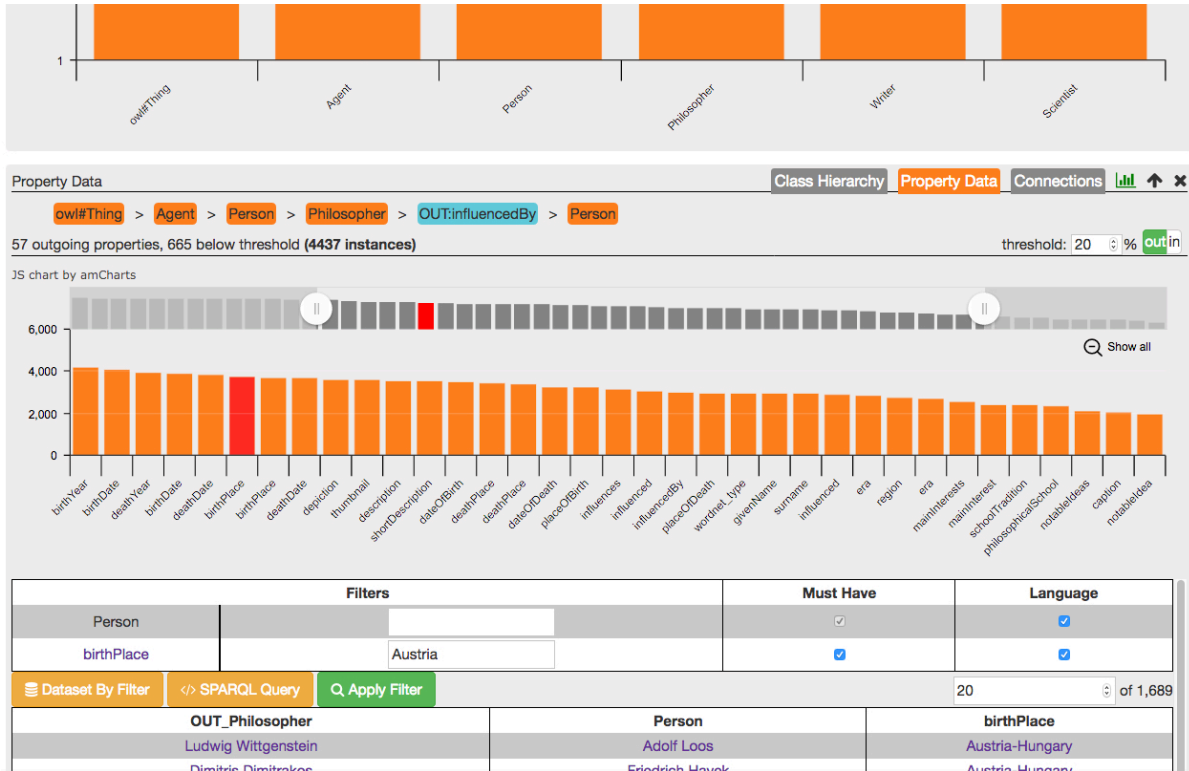


Figure 2: Screenshot of two exploration panes over DBpedia (upper is partially visible). Lower pane shows property data about persons who have influenced philosophers.

subjects (instances of class owl:Thing) into subclasses. Each bar matches a specific subclass, with a height proportional to its number of instances. The bars are sorted by decreasing height. Hovering over a bar opens a pop-up box with basic information, for instance, DBpedia’s Agent class has more than 2 million instances, 5 direct subclasses, and 275 subclasses in total.

To support visualization of large charts, a widget allows to control the visible part of the chart. Class navigation is done by clicking a bar, which opens a new pane under the current one. For example, navigation to type Philosopher involves opening three panes: Agent → Person → Philosopher. Alternatively, an autocomplete search box for class types may be used, in cases where top-down class navigation is less intuitive.

Property and object expansions. A pane has a second tab with a *property chart*—the result of a property expansion. Figure 2 shows a property chart for Person subjects who have influenced philosophers. Here, a bar represents instances that share a specific property. (Switching to an *incoming* chart shows incoming properties, as explained in Section 2.) Bars are sorted by *coverage*—the percentage of instances that feature the property. The number of possible properties may be very large, and thus, ELINDA filters out properties with a coverage lower than a threshold (defaults to 20% and adjustable by the user). In the figure, only 57 properties out of 722 possible properties are shown.

The property chart in the figure was derived *after* applying an object expansion to a previous pane of type Philosopher. In that previous pane (partially shown in Figure 2), an influencedBy property was selected in its property chart. Then, via a third *object chart*, ELINDA was asked to open a new pane (current pane) with instances of type Person connected to philosophers through

the influencedBy property. This allows the user to further explore *only* persons who have influenced philosophers.

A user interested in looking into the details of the dataset, may use the *data table* that appears below the chart. Upon selection of properties (bars) in the chart, columns are added to the table and filled-in with values fetched from the dataset. The SPARQL query used to generate the data table may be retrieved by the user for future consumption. *Data filters* attached to table columns may restrict the displayed data. In Figure 2, only persons born in *Austria* (and have influenced philosophers) are presented.

4 SYSTEM ARCHITECTURE

The architecture design of ELINDA is driven primarily by responsiveness, aiming for bar chart expansions to occur instantly. This is challenging, since some of the queries that are submitted to the endpoint require an execution time of up to several minutes on a Virtuoso endpoint. This is mostly because Virtuoso utilizes traditional join algorithms, which generate up to billions of intermediate results that are not part of the final result.

ELINDA expansion queries retrieve the subject distribution between subclasses or properties of a given class. Retrieving all subjects of a class incurs a reachability query that retrieves the subjects of all transitive subclasses. For example, when applied to class Thing, the query retrieves all of its subjects, including those of direct and indirect subclasses. The subjects are later grouped by each Thing subclass or property and counted distinctly.

To provide the required responsiveness, we built a novel exploration query engine called ELINDA-QE. Our query engine is based on *Cached Trie Join* (CTJ) [5], a worst case optimal join algorithm. CTJ generates only partial intermediate results that will accelerate the join query. CTJ shows orders of magnitude

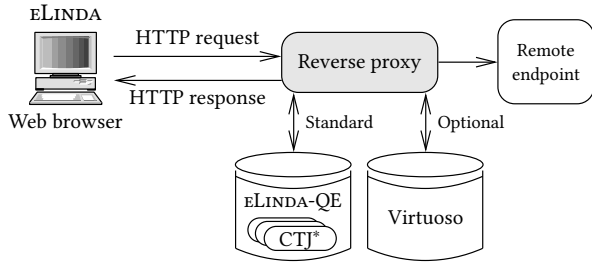


Figure 3: Basic system architecture of eLINDA.

speedup over traditional approaches on graph workloads. Yet, CTJ supports only equi-join queries, and COUNT and SUM aggregations. To support eLINDA expansions, we extended CTJ and refer to the extended algorithm as CTJ*. First, our CTJ* can compute reachability queries. Second, CTJ aggregations were extended to support group aggregations. Queries containing outer joins, such as queries containing the OPTIONAL clause, are not supported by CTJ*. Such queries are offloaded to a Virtuoso endpoint. We plan to add outer join support to CTJ* in future work. Different orders of (s, p, o) indexes are maintained and used by CTJ*, similar to the indexes managed by Virtuoso.

Figure 4 shows the runtime of the slowest and most commonly used queries by eLINDA on an Ubuntu server with 16 cores and 128GB RAM. These queries construct the bar charts of the outgoing and incoming property expansions and the subclasses expansions in the first levels. The Virtuoso endpoint is configured to utilize all available memory if needed. For all the queries in Figure 4, the eLINDA-QE is 1–2 orders of magnitude faster than Virtuoso. For example, the runtime of class Thing property expansions on the Virtuoso SPARQL endpoint is 630 and 96 seconds for the incoming and outgoing bar charts, respectively. On the eLINDA engine, the runtime is 11 and 3 seconds, respectively. The speedups are consistent with other heavy queries we tested.

eLINDA remote mode can work with any Virtuoso SPARQL endpoint. Remote mode incorporates two methods to provide effective latency for a user interface. First, an incremental evaluation is being applied. eLINDA builds the chart of an expansion by computing it on the first N triples in the RDF graph. It then continues to compute the query on the next N triples and aggregates the results in the frontend. It continues for k steps, or until the full chart is computed. The parameters N and k are determined by an administrator configuration. Second, caching is used to reduce the latency. These methods allow eLINDA to quickly present information to the user that otherwise will take minutes or be rejected by the endpoint for running too long.

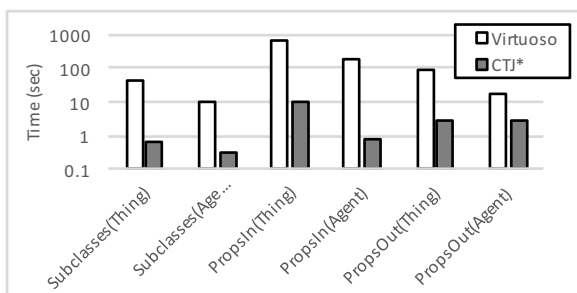


Figure 4: Running times of property and subclass expansions on different engines (log scale).

Figure 3 depicts the architecture of eLINDA. The frontend is implemented as a Web page that communicates with the server via AJAX. Expansion queries are sent to eLINDA engine (running CTJ*) while OPTIONAL queries are offloaded to a Virtuoso endpoint. The eLINDA engine is a Web server developed in C++17, incorporates a multithreaded Web API for query evaluation, and uses OpenMP for fine-grained parallelism in CTJ*. In remote mode, all queries are sent to the remote endpoint and are cached either in the reverse proxy or in the browser.

5 DEMONSTRATION SCENARIOS

During the demonstration, participants will explore several RDF datasets such as DBpedia and LinkedGeoData with eLINDA. Several kinds of explorations will be exercised.

The first kind will tackle the task of understanding a large and unfamiliar dataset. The participants will examine the bar chart showing the first-level classes of the dataset (subclass expansion). They will be presented with key statistics that may be inferred from the chart, such as the three largest classes, the number of instances these classes have, and the number of their direct and indirect subclasses.

In the second scenario, the participants will analyze the property chart of the largest class in the dataset (property expansion). They will examine the twenty most significant properties, then select a few of them and see their values appear in the data table. Selected properties will be added with filters, and the data presented in the table will be reduced. The participants will adjust the default coverage threshold to 50% and see the number of presented properties decreasing. Similarly, incoming properties will be explored. Additional scenarios will look into sophisticated exploration paths such as “the types of people that influenced philosophers,” “cities where scientists were born,” and “spouses of former US presidents.” These scenarios will involve opening several charts in sequence to achieve the desired goal.

Another scenario will demonstrate the performance issue elaborated in Section 4. The participants will be presented with several explorations that entail heavy queries with the discussed solutions turned on and off. This demonstration will include working with the described eLINDA engine, as well as working in remote mode where standard Virtuoso SPARQL endpoints are used “as is.” In the remote mode, incremental evaluation of queries and the use of caching will be illustrated.

The last scenario will demonstrate how eLINDA can be used to detect *erroneous* data such as “people who are indicated to be born in resources of type food.”

REFERENCES

- [1] C. Bizer, J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, and S. Hellmann. DBpedia - a crystallization point for the web of data. *Web Semant.*, 7(3):154–165, Sept. 2009.
- [2] A.-S. Dadzie and M. Rowe. Approaches to visualising linked data: A survey. *Semantic Web*, 2(2):89–124, 2011.
- [3] S. M. Falconer, C. Callendar, and M.-A. D. Storey. A visualization service for the semantic web. In *EKAW*, volume 6317 of *LNCIS*, pages 554–564. Springer, 2010.
- [4] W. Hop, S. de Ridder, F. Frasnier, and F. Hogenboom. Using hierarchical edge bundles to visualize complex ontologies in GLOW. In *SAC*, pages 304–311. ACM, 2012.
- [5] O. Kalinsky, Y. Etsion, and B. Kimelfeld. Flexible caching in trie joins. In *EDBT*, pages 282–293, 2017.
- [6] B. Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *VL*, pages 336–343, 1996.
- [7] G. Tummarello, R. Cyganiak, M. Catasta, S. Danielczyk, R. Delbru, and S. Decker. Sig.ma: Live views on the web of data. *J. Web Sem.*, 8(4):355–364, 2010.
- [8] M. Weise, S. Lohmann, and F. Haag. LD-VOWL: extracting and visualizing schema information for linked data endpoints. In *VOILA*, volume 1704 of *CEUR-WS*, pages 120–127. CEUR-WS.org, 2016.

FAIMUSS: Flexible Data Transformation to RDF from Multiple Streaming Sources

Georgios M. Santipantakis
 Department of Digital Systems
 University of Piraeus
 18534 Piraeus, Greece
 gsant@unipi.gr

Apostolos Glenis
 Department of Digital Systems
 University of Piraeus
 18534 Piraeus, Greece
 apostglen46@gmail.com

Nikolaos Kalaitzian
 Department of Digital Systems
 University of Piraeus
 18534 Piraeus, Greece
 nikoskalai@gmail.com

Akrivi Vlachou
 Department of Digital Systems
 University of Piraeus
 18534 Piraeus, Greece
 avlachou@aueb.gr

Christos Doulkeridis
 Department of Digital Systems
 University of Piraeus
 18534 Piraeus, Greece
 cdoulk@unipi.gr

George A. Vouros
 Department of Digital Systems
 University of Piraeus
 18534 Piraeus, Greece
 georgev@unipi.gr

ABSTRACT

In this paper, we present *FAIMUSS* a tool for data transformation from a wide variety of heterogeneous streaming and archival sources to RDF. This is a typical situation in the analysis of mobility data, such as maritime and aviation, where streaming position data of moving objects need to be associated with static information (such as crossing sectors, protected geographical areas, weather, etc.) in order to provide semantically enriched trajectories. *FAIMUSS* is designed to perform “near-to-the-sources” integration, by interaction between a streaming source and an archival source, thus generating linked RDF graph fragments. Most of the existing approaches operate either on streaming or on static sources, thus fail to address our problem setting. In addition, *FAIMUSS* supports reusable user-defined functions that are applied to input data and achieve the desired data transformations and cleaning. We demonstrate our prototype by using data from the maritime and aviation domains.

1 INTRODUCTION

The Resource Description Framework (RDF) enables the description of physical entities as resources, in favor of data interoperability, integration and exchange. As a result, a wide range of solutions exists for converting a data source to RDF, based on a schema specified by RDFS or OWL profiles. For instance, R2RML [5] is a mapping language that translates SPARQL queries to SQL (and vice-versa). Ontology Based Data Access (OBDA) [2, 3] also focuses on relational data sources, but again requires advanced knowledge to define the (usually complicated) mappings between the data source and the ontology schema. SPARQL-Generate [10] provides an extension of SPARQL 1.1 that allows generation of RDF fragments from documents. Solutions tailored for specific RDF stores also exist, for example Virtuoso Cartridge¹, however they are tied to a particular product.

Moreover, the amount of streaming data sources has increased rapidly in recent years, and such sources pose new challenges for data integration [13]. Apart from the obvious performance challenge raised by high stream rate, streaming data are typically

semi-structured, and contain noisy data. As such, the problem of integrating streaming data sources is still open, and there is a lack of flexible tools that transform streaming data to RDF, integrate it with external sources, and are easily configurable to new sources.

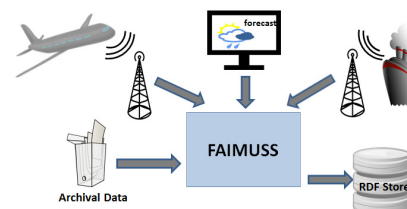


Figure 1: *FAIMUSS* consumes streaming and archival sources and produces (linked) RDF data.

Our work is motivated by the need to produce enriched representations of moving object trajectories expressed in RDF, by ingesting streaming surveillance data in real-time and associating it with other sources, both streaming and archival. Figure 1 illustrates this process, focusing on mobility data in maritime [4] and aviation domains recording positions of moving objects, such as vessels and aircrafts. However, this is simply for demonstration reasons, as our work is readily applicable to any other category of streaming data, including social data consumed through an API (e.g., Twitter API).

In this paper, we present *FAIMUSS* (Flexible dAta TransformatIon to RDF from Multiple Streaming Sources), a flexible, user-friendly system for end-to-end data transformation of streaming data to RDF and integration with external sources. Given an ontology, *FAIMUSS* converts streaming data (but also other data sources) into RDF triples, which are also integrated with other archival data sources. We employ the datAcron ontology [12] for representing trajectories at multiple levels of analysis. However, the system imposes no constraints on the use of a specific ontology, while it supports a wide range of input source data formats. Salient features of *FAIMUSS* include its flexibility and user-friendliness: the process of triple generation is determined by a *Graph Template* that is easily edited by the user with the support of a rich editor. Furthermore, *FAIMUSS* supports “near-to-the-sources” integration, by generating *linked* RDF graph fragments during the process of data transformation. Finally, the implementation of *FAIMUSS* also addresses scalability issues and aims at high performance.

¹<https://virtuoso.openlinksw.com/dataspace/doc/dav/wiki/Main/VirtProgrammerGuideRDFCartridge>

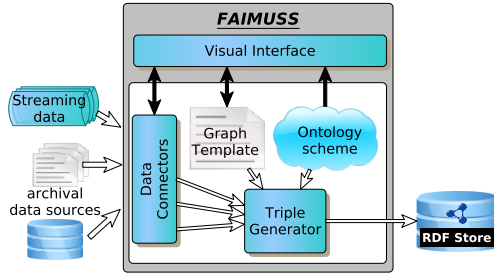


Figure 2: System architecture of FAIMUSS. White arrows indicate data flows and black arrows indicate user interactions.

2 SYSTEM ARCHITECTURE

In this section, we describe the system architecture that consumes data from a variety of data sources and outputs RDF graph fragments. This process is designed in a generic fashion, so as to accomplish multiple objectives, including flexibility, extensibility and scalability. FAIMUSS comprises the following main components, a) the *Data Connector*, b) the *Triple Generator*, and c) the *Visual Interface*. Figure 2 illustrates the overall system and the individual components and their interactions, which will be described in detail. The prototype system has been implemented in Oracle Java 8 (64-bit) and is platform-independent.

2.1 Data Connector

The *Data Connector* component implements the functions that accept data from an individual data source. Moreover, it performs data conversion on values of specific fields as provided by the source and basic data cleaning operations. As the data sources can vary significantly in terms of representation, size, rate of data access, and noise, FAIMUSS provides a Data Connector for each type of source. The implemented system currently supports data consumption from a wide range of sources/formats, such as: a) CSV format, b) direct access to databases, c) JSON messages, d) XML files, e) METAR/SPECI weather reports from offline/continuous feeds from online services, f) binary (GRIB2) files for weather reports, g) SPARQL endpoints to online open data, h) ESRI shape-files to convert information about spatial object to entities and relations of the ontology, and i) proprietary formats of streaming. The list of supported data formats can be easily extended, to support the inclusion of any other data format required in the future.

As an abstraction of the data source in hand, the Data Connector considers all data sources as streams, i.e. it consumes data record-by-record (or tuple-by-tuple), to be processed with minimal latency. This data access model makes no distinction on the nature of data (e.g. archival data or streaming data). In addition, it minimizes the memory footprint of Data Connectors, and enables scalability and parallelization as a multi-threaded process, both for a single source and across sources.

2.2 Triple Generator

The role of *Triple Generator* is to consume the data provided by the Data Connector and generate the corresponding set of triples. This procedure depends on configuration files providing a Graph Template G_T , and a vector of variable names V . The vector V contains the variables that appear in the Graph Template, and binds the values of the input record to the variables in G_T . The

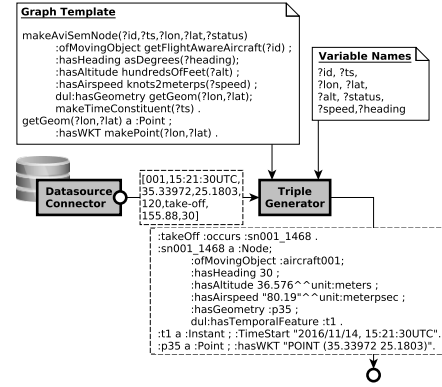


Figure 3: A Graph Template and Variable Vector example in Triple Generator.

Graph Template consists of a set of *triple patterns*, i.e. any of the three elements in the triple (subject, predicate, object) can be replaced by a variable. The Graph Template differs from the standard RDF Graph Pattern, in that variables can be used as arguments in functions, to compute dynamic values at runtime. Furthermore, the implementation of *Triple Generator* is parallelized in a multi-threaded process, due to the record-by-record conversion to triples.

Figure 3 illustrates the application of a Graph Template on a single record from a stream of aircraft surveillance data to generate the corresponding triples. Specifically, the Triple Generator consumes the data provided from the Data Connector and constructs a new resource (`:sn001_1468`) of type `:Node`, based on the Graph Template depicted at the top of the figure. This new resource is the spatio-temporal representation of the moving object (aircraft). The Triple Generator also relates the `:sn001_1468` to the resource representing an aircraft (using the property `:ofMovingObject`), as well as information regarding the status, the altitude and position of the moving object.

Figure 4 shows another example of data transformation where two data sources are processed and integrated, by having two instances of Triple Generator interact with each other. The positioning data connector provides surveillance data of vessels to a Triple Generator instance, similar to the previous example. The GRIB connector accesses binary files that contain weather forecasts and can extract the related weather attributes for a given spatio-temporal position. As soon as a new position is received, a request is made to the Triple Generator instance that retrieves the weather information and provides it in RDF representation based on the respective Graph Template. This allows data transformation of selected weather information, namely this corresponding to the area defined by the positioning information. More interestingly, the weather Triple Generator returns the URI of the weather condition to the positioning Triple Generator, which can then create the `:hasWeatherCondition` property and associate the position with the weather. This is an example of lightweight, “near-to-the-sources” integration.

2.3 Visual Interface

The *Visual Interface* enables the configuration of Data Connector and Triple Generator, the preview of input data, the visualization of output triples on a map, and output validation. Specifically, the user can select the ontology to be used, which will provide

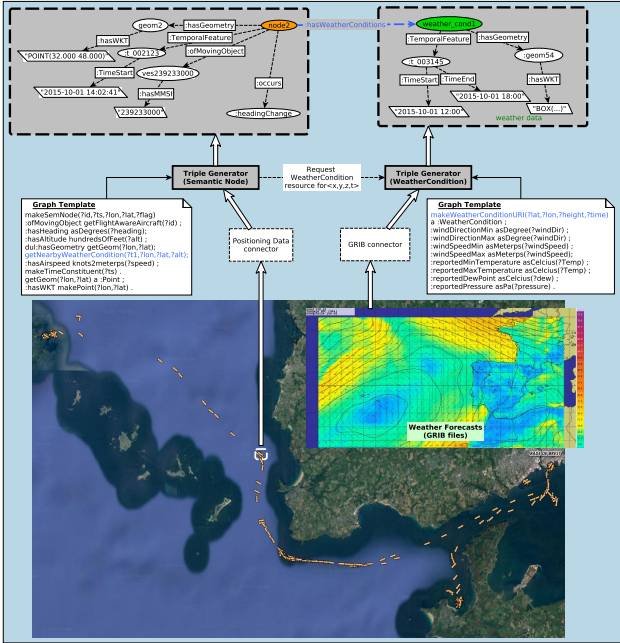


Figure 4: Example of data transformation and “near-to-the-sources” integration (maritime domain).

the vocabulary for the Triple Generator. Obviously, the ontology should be taken into consideration when editing a Graph Template, to guarantee consistency of the generated triples.

Figures 5 and 6 provide screenshots of the Visual Interface using data from the aviation domain. As illustrated in Figure 5(a), the user can select and configure the data source that will be processed (e.g., specify the IP and port of remote endpoints, or folder/file for local access), and the type of Data Connectors that should be used for consuming the data. A sample of the data available in the configured data source, for visual inspection. The user can also select the fields of the input data source that should be converted to triples, and specify the *Variables Vector* to be used in the Graph Template. Figure 5(b) shows the editor for composing the Graph Template. On the right side, the set of available functions is provided, from which the user can insert in the editor by a double click. Also, the Variables corresponding to the input source are also available. The editor uses different font color for functions and variables to improve readability. In addition, import (export) of Graph Templates from (to) disk is supported. It should also be mentioned that automatic validation of a sample (or the entire output) of triples generated w.r.t. the provided Graph Template is supported. Figure 6 depicts a map-based interface provided by FAIMUSS, which enables the illustration of spatial and spatio-temporal RDF data. In this example, it depicts trajectories of moving objects (aircrafts), which correspond to RDF data generated by our system.

3 DEMONSTRATION SCENARIOS

We have used FAIMUSS in two domains, maritime and aviation, thus showing the generic nature of our system and its applicability in different domains. The selection of these domains originates from our involvement in the H2020 research project

datAcron (<http://datacron-project.eu/>), where one of the objectives is to *integrate heterogeneous and voluminous archival data with streaming data sources*.

Scenario 1: Flexible Integration of New Sources. We intend to show in the demo that a new streaming data source can be added to FAIMUSS, after a small set of steps that can be performed by selections from the GUI. The aim is to demonstrate the ease of use of FAIMUSS in practice. In more detail, the streaming data source is going to be surveillance data from vessels travelling in the Mediterranean Sea. Each record in the stream contains the spatio-temporal position of a moving object, along with its unique identifier. From the GUI of FAIMUSS, we are going to input the necessary information for establishing a connection (IP address, port, and authentication details). Thereafter, a Graph Template is going to be specified from an editor of the GUI, which guides the Triple Generator and determines the transformation of stream records into RDF triples. As already mentioned, the editor facilitates the composition of Graph Templates even for moderately familiar users, not only by loading existing Graph Templates that can be modified, but also by providing access to lists of available functions and variables that generate Graph Template code in the editor.

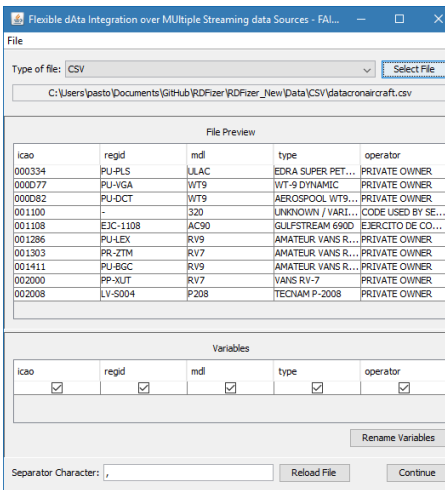
Scenario 2: Trajectory Enrichment. We will show in the demo how we can generate enriched trajectories from moving objects’ positions, where the positional information can be integrated with arbitrary data sources. Continuing scenario 1, the spatio-temporal position of each vessel is going to be integrated with weather conditions. Weather conditions are available as forecasts for a temporal period (e.g. three hours) in the form of large binary files (GRIB2 format), and mainly define 3-dimensional cells that contain multiple variables describing weather condition (e.g., temperature, humidity, pressure, etc.). In this scenario, upon receipt of a spatio-temporal position of a moving object, the Triple Generator instance responsible for surveillance data is going to produce RDF triples. By communication with a Triple Generator instance for weather data, which produces weather-related triples, the resulting positional RDF data is linked with weather information. In this way, we generate enriched trajectories of moving objects represented in RDF, which will be illustrated on a map for visual inspection, and provide additional (weather-related) information for each spatio-temporal position of a moving object.

4 RELATED WORK

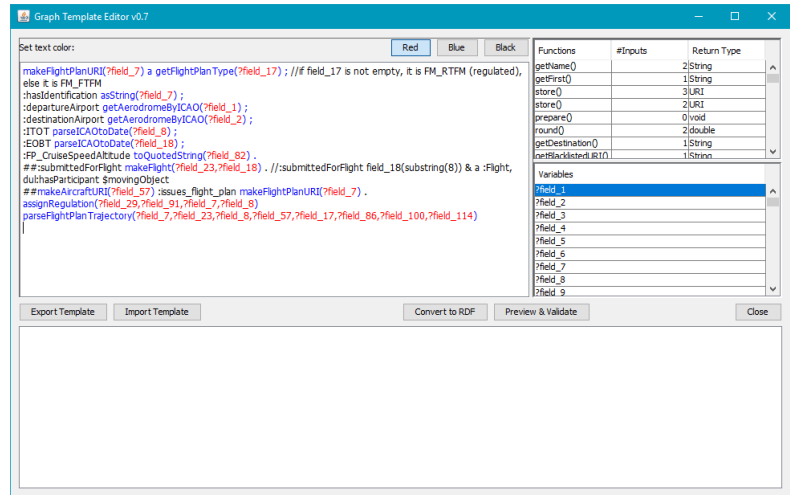
A wide list of data transformation and conversion tools exists, and such tools² have been evaluated during the design of our system, however these are mostly ad-hoc solutions tailored for converting archival data in specific file formats. For example, Omnidator [1] converts any CSV or HTML online file into RDF triples. GeoTriples [8] employs automatically generated R2RML mappings given a source and a configuration, but demands the use of a relational database, which is not applicable for streams. SPARQL-Generate [10] provides an extension of SPARQL 1.1 that allows generation of RDF fragments from documents.

Data integration over streaming data poses new challenges compared to traditional data integration [13]. The Graph of Things [9] targets an IoT setting where many sources provide data for integration and querying, and supports spatial and temporal data, but it is not optimized for mobility data. Also related to our work is StreamLoader [11] which provides a user-friendly,

²<https://www.w3.org/wiki/ConverterToRdf>



(a) Configuration of Data Connector



(b) Editor for Graph Templates

Figure 5: Screenshots of the Visual Interface of FAIMUSS (aviation domain).

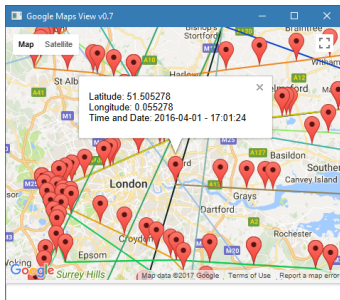


Figure 6: Map-based visualization.

web-based environment for ETL (Extract-Transform-Load) of streaming data from heterogeneous sensors. However, it deviates from our objective, namely to provide representations of streaming data in RDF, thus allowing linking with other external (web) sources. Moreover, it does not explicitly address the domain of moving objects and trajectories thereof, which is the motivation of our work. Only recently, Optique [6, 7] proposes an approach for integration of streaming with static relational data. Again, this problem is much narrower than the one addressed by our work, since we make no assumptions on the availability of a relational database nor do we impose such requirements. None of the existing approaches targets streaming mobility data of vessels or aircrafts explicitly. Moreover, our work goes one step further, by introducing a system for flexible data transformation to RDF, with a particular focus on mobility data, also supporting linking of generated RDF graph fragments.

5 CONCLUSIONS AND FUTURE WORK

In this paper, we present the FAIMUSS system for flexible data transformation of streaming and archival data sources to RDF. Despite the generic and extensible design of our system, the focus of our attention is on moving objects and integrating their spatio-temporal positions with a variety of heterogeneous data sources, including weather conditions, spatial areas of interest, as well as external databases. In our future work, we intend to

fully parallelize our system, thus providing a scalable solution to the problem of data integration from different data sources.

ACKNOWLEDGMENTS

This work has been supported by project datAcron, which has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 687591.

REFERENCES

- [1] Omnidator. <http://omnidator.appspot.com/>.
- [2] C. Bizer and A. Seaborne. D2RQ-treating non-RDF databases as virtual RDF graphs. In *Proc. of ISWC*, 2004.
- [3] D. Calvanese, B. Cogrel, S. Komla-Ebri, R. Kontchakov, D. Lanti, M. Rezk, M. Rodriguez-Muro, and G. Xiao. Ontop: Answering SPARQL queries over relational databases. *Semantic Web*, pages 1–17, 2016.
- [4] C. Claramunt, C. Ray, E. Camossi, A. Jousseime, M. Hadzagic, G. L. Andrienko, N. V. Andrienko, Y. Theodoridis, G. A. Vouros, and L. Salmon. Maritime data integration and analysis: recent progress and research challenges. In *Proc. of EDBT*, pages 192–197, 2017.
- [5] S. Das, S. Sundara, and R. Cyganiak. R2RML: RDB to RDF mapping language. <https://www.w3.org/TR/r2rml/>, 2012.
- [6] E. Kharlamov, S. Brandt, E. Jiménez-Ruiz, Y. Kotidis, S. Lamparter, T. Mailis, C. Neuenstadt, Ö. L. Özçep, C. Pinkel, C. Svingos, D. Zheleznyakov, I. Horrocks, Y. E. Ioannidis, and R. Möller. Ontology-based integration of streaming and static relational data with optique. In *Proc. of SIGMOD*, pages 2109–2112, 2016.
- [7] E. Kharlamov, Y. Kotidis, T. Mailis, C. Neuenstadt, C. Nikolaou, Ö. L. Özçep, C. Svingos, D. Zheleznyakov, S. Brandt, I. Horrocks, Y. E. Ioannidis, S. Lamparter, and R. Möller. Towards analytics aware ontology based access to static and streaming data. In *Proc. of ISWC*, pages 344–362, 2016.
- [8] K. Kyzirakos, I. Vlachopoulos, D. Savva, S. Manegold, and M. Koubarakis. GeoTriples: a tool for publishing geospatial data as RDF graphs using R2RML mappings. In *TC/SSN@ISWC*, pages 33–44, 2014.
- [9] D. Le-Phuoc, H. N. M. Quoc, H. N. Quoc, T. T. Nhat, and M. Hauswirth. The Graph of Things: A step towards the Live Knowledge Graph of connected things. *Web Semantics: Science, Services and Agents on the World Wide Web*, 37:25–35, 2016.
- [10] M. Lefrançois, A. Zimmermann, and N. BAKERALLY. A SPARQL extension for generating RDF from heterogeneous formats. In *Proc. of ESWC*, pages 35–50, 2017.
- [11] M. Mesiti, L. Ferrari, S. Valtolina, G. Licari, G. L. Galliani, M. Dao, and K. Zettsu. StreamLoader: An Event-Driven ETL System for the On-line Processing of Heterogeneous Sensor Data. In *Proc. of EDBT*, pages 628–631, 2016.
- [12] G. Santipantakis, G. Vouros, C. Doukeridis, A. Vlachou, G. Andrienko, N. Andrienko, G. Fuchs, J. M. C. Garcia, and M. G. Martinez. Specification of semantic trajectories supporting data transformations for analytics: The datAcron ontology. In *Proc. of Semantics*, 2017.
- [13] N. Tatbul. Streaming data integration: Challenges and opportunities. In *Proc. of ICDE*, pages 155–158, 2010.

SAMUEL: A Sharing-based Approach to processing Multiple SPARQL Queries with MapReduce*

InA Kim
 Dep't of Computer Engineering
 Chungnam National University
 Daejeon, 34134, Korea
 dodary0214@gmail.com

Kyong-Ha Lee[†]
 Korea Institute of Science and
 Technology Information
 Daejeon, 34141, Korea
 bart7449@gmail.com

Kyu-Chul Lee
 Dep't of Computer Engineering
 Chungnam National University
 Daejeon, 34134, Korea
 kclee@cnu.ac.kr

ABSTRACT

The volume of RDF data is now growing tremendously. It is thus considered prudent to store and process massive RDF data with distributed SPARQL engines instead of relying on a single-machine system. Many sophisticated index and partitioning schemes have also been proposed to support SPARQL query evaluations. However, existing SPARQL engines have mainly followed *one-at-a-time* scheme so that query evaluation is focused only on processing each query separately. We showcase *SAMUEL*, a distributed SPARQL engine that simultaneously evaluates many SPARQL queries for a massive RDF dataset with MapReduce. *SAMUEL* provides an efficient optimization algorithm to evaluate many SPARQL queries simultaneously in a shared and balanced way. Extensive experiments present that without any sophisticated partitioning or index mechanisms, our approach significantly outperforms other MapReduce-based SPARQL engines as well as an *ad-hoc* query engine equipped with various indexes and partitioning tools for evaluating multiple SPARQL queries.

1 INTRODUCTION

The Resource Description Framework (RDF) is a versatile graph data model that enables users to express facts and their relationships in the form of triples $\langle \textit{Subject}, \textit{Predicate}, \textit{Object} \rangle$ where a predicate (*P*) expresses a relationship between a subject (*S*) and an object (*O*) [2]. Many knowledge bases are now defined in the RDF format, e.g., DBpedia [16], Bio2RDF [6] and UniProt [7] and they shape a very large set of graphs having millions of triples interlinked each other and are queried by using SPARQL query language [1]. It has been a major challenge to find subgraph patterns described in given SPARQL queries from a massive set of RDF graphs with supporting both efficiency and scalability. To address the issue, numerous SPARQL query engines have been devised based on MapReduce [15] or their proprietary distributed architectures [3]. Meanwhile, multiple SPARQL queries often need to be evaluated together as the queries can be prepared before runtime in some scenarios [13, 20]. Motivated by these facts and our previous work on XML data [8], we devise a MapReduce-based SPARQL engine called *SAMUEL*, which simultaneously evaluates many SPARQL queries in a *shared* and *balanced* way. Major features of *SAMUEL* are as follows :

Support of parallel multi-SPARQL query processing

SAMUEL provides an efficient means to process a massive set of

RDF data in parallel. It does not require any sophisticated partitioning or index mechanisms for SPARQL query evaluation. Nonetheless, *SAMUEL* easily outperforms other MapReduce-based distributed SPARQL engines by simultaneously evaluating multiple SPARQL queries with a short series of MapReduce jobs.

Sharing input scans and intermediate results

SAMUEL enables computing nodes to share input scans and their intermediate results with each other. While joining RDF triples for evaluating queries, a group of join operations assigned to each reducer share their input and intermediate results associated with distinct subquery patterns that multiple queries commonly contain. Consequently, it saves many I/Os by removing many redundant subquery matchings while evaluating queries.

Runtime load balancing and multi-query optimization

In a distributed system, a straggling task delays overall job execution. This problem deteriorates with the use of MapReduce since MapReduce typically enforces barrier synchronization between Map and Reduce tasks. MapReduce's native runtime scheduling algorithm also proved to be inefficient especially in the reduce stage [12, 15]. To address the issue, *SAMUEL* rather exploits *dynamic shuffling* scheme that balances workloads across reducers at each MapReduce job in accordance with the cardinality of RDF triples that each reducer consumes for joining. It decomposes a given set of SPARQL queries into distinct triple patterns and then *gradually* builds a bushy query plan tree that covers all RDF join operations required to evaluate the given SPARQL queries. Since processing join operations at each level in the query plan tree requires a single M/R job, *SAMUEL* always tries to build a bushy plan tree with the lowest possible height. Then, it partitions the join operations at each level into n groups and then assigns them to n reducers. To balance workloads across reducers that actually perform join operations, *SAMUEL* computes the cost of each join operation at each level of the plan tree before actual joining. It then assigns join operations into reducers at each level such that every reducer has the same overall cost of join operations and the lowest communication cost by avoiding redundant RDF triples being transferred to multiple reducers so far as possible.

The rest of this paper is organized as follows. Section 2 introduces previous studies directly related to our work. Section 3 describes how we evaluate multiple SPARQL queries together at a time and our system architecture that provides workload balancing as well as multi-SPARQL query processing. Section 4 presents our demonstration scenario including the major results of performance evaluations.

2 RELATED WORK

Numerous distributed SPARQL engines have been devised to store RDF data and to evaluate SPARQL queries so far [3, 9, 11, 14, 19, 21, 22]. Readers are referred to a recent survey on distributed SPARQL query engines [3]. These systems fall into

*This work is equally supported by KISTI and NRF grant (No. NRF-2015R1A2A2A01004879) funded by the Korea government.

[†]Corresponding author

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

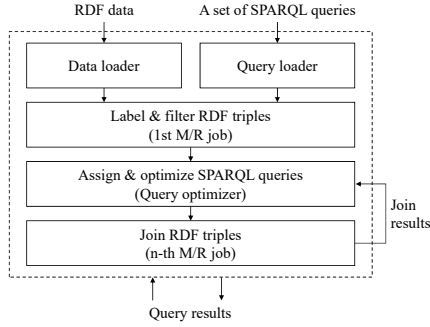


Figure 1: System architecture

two categories in the aspect which framework these systems rely on : (i) general-purpose framework such as MapReduce and (ii) specialized RDF systems. In this paper, we only deal with MapReduce-based RDF engines due to the limit of space.

MapReduce-based RDF engines

MapReduce is a popular parallel processing tool that provides high scalability as well as simple abstraction. Therefore, many studies have been done with MapReduce although it has some inherent limitations [15]. It is noteworthy that most M/R-based RDF engines do not well utilize index mechanisms since MapReduce is originally devised for *batch processing* rather than *ad hoc queries*. Indexes are considered inefficient for batch jobs due to their expensive building cost only for one-time use.

SHARD [21] is a distributed RDF engine built with MapReduce. All the RDF triples are stored in a single file in HDFS and RDF triples are hash-partitioned across nodes. SPARQL queries are evaluated as M/R job iterations. A subquery pattern is evaluated within a M/R job and its results are transferred to a subsequent M/R job. SHARD does not utilize any indexing scheme so that it needs to scan the entire dataset for query evaluation. HadoopRDF [11] is another RDF engine built with MapReduce. It partitions RDF triples into multiple files in the way that each file contains all the triples that have the same predicate. It also locates non-duplicate binary joins together into a M/R job to minimize MR job iterations. SHAPE [14] also uses MapReduce but it uses semantic hash partitioning scheme to group vertices on the basis of URI hierarchy for improving data locality. However, these systems suffer from workload imbalance if a few triple patterns dominates overall data distribution. H2RDF+ [19] is a distributed engine based on both MapReduce and HBase, the open sourced version of BigTable. It materializes combinations of RDF triples and store them into HBase tables in order to utilize some features of HBase, *e.g.*, sorted keys and range-partitioned tables based on the keys. However, H2RDF still join RDF triples with M/R job iterations thus a complex queries can expand the iterations. CliqueSquare [9] partitions RDF triple in three ways, by hashing them on the basis of subject, predicate and object for increasing data locality. It exploits the data replication feature of HDFS to locally process all first-level joins on each node. It also tries to minimize the number of M/R job iterations for RDF joins with a shallow query plan tree and multi-way join operations.

Multi-query optimizations

Multi-query optimization on MapReduce is regarded as an extended version of the classical *job-shop problem* [5] that has a long history. A few studies related to multi-query optimization on MapReduce have been reported in the literature [18, 23] in the context of relational processing. They provide generalized grouping techniques that merge multiple jobs into a single job

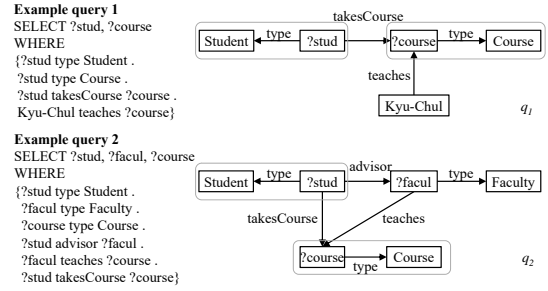


Figure 2: Example of common subquery patterns in two SPARQL queries

thereby enabling the merged jobs to share input scans and common mapped outputs. Some studies address the issue in the context of SPARQL queries [13, 20]. Multi-query optimization for SPARQL is also proven to be *NP-hard* so that they rather propose heuristic algorithms that partition a set of queries into groups such that queries in each group can be optimized together. However, their algorithms [13, 20] are focused only on working on a single-machine engine rather than a distributed environment such as MapReduce. Therefore, all the approaches are hard to be directly applied to distributed SPARQL engines. On the contrary, we focus on multi-query optimization on MapReduce-based distributed SPARQL engines. To the best of our knowledge, this is the first work that provides a solution for the multi-query optimization problem in MapReduce-based SPARQL engines.

3 MULTI SPARQL QUERY PROCESSING

SAMUEL performs its query evaluation in three phases: (i) preprocessing phase, (ii) labeling & filtering phase, and (iii) iterative joining phase (see Fig. 1). In the preprocessing phase, RDF data and a set of SPARQL queries are loaded on HDFS. In our system RDF triples are simply stored in a single file where each triple is recorded as a single line as it is, except that redundant URIs and labels are substituted by unique IDs for saving storage volume and I/Os. Note that we do not use any partitioning or index mechanisms for RDF data since we only show the effect of our approach, distinguished from the benefits that we can get with the mechanisms. Query loader decomposes SPARQL queries into a set of distinct triple patterns, and then associates each triple pattern with a set of queries that contains the triple pattern like Fig. 2. It also builds an in-memory radix tree where a tree node at each level represents a unique *S*, *P*, and *O* of triple patterns, respectively. With the radix tree, SAMUEL rapidly finds triple patterns matched to an input RDF triple while query evaluation.

The labeling & filtering phase is implemented with a single M/R job. In the phase, RDF triples are labeled with IDs for their corresponding triple patterns by traversing the radix tree and also are filtered out if they have no corresponding triple patterns. This work has an analogy to filtering stream data. Since reducers aggregate RDF triples that has the same pattern, it allows us to easily compute the cardinality of each triple pattern in the phase.

Based on the cardinality information, our query optimizer builds a global query plan tree that has the lowest possible height to minimize the number of M/R job iterations for joining RDF triples. Redundant join operations are removed as each join operation is shared by multiple queries. When assigning binary RDF join operations to reducers, we consider both join cost and transmission cost to minimize and balance workloads across reducers. It is achieved by summing the cost of every RDF join operation

Dataset	LUBM-100M	WatDiv-1M	WatDiv-10M	WatDiv-100M	WatDiv-1B
# of triples in total	138,280,374	1,098,468	10,989,614	109,795,305	1,098,717,244
# of distinct triples	133,573,856	1,085,817	10,906,204	109,051,965	1,091,667,092
# of distinct patterns matched to queries (1,000 queries for WatDiv and 14 queries for LUBM)	23	361	355	358	355
Data size(GB)	23.02	0.14	1.45	14.63	148.23

Table 1: Statistics of RDF Datasets

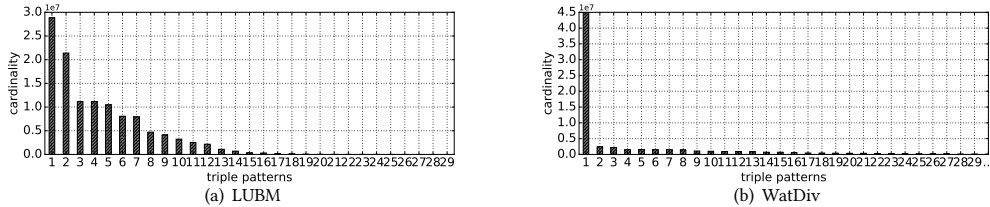


Figure 3: Data skewness in two datasets

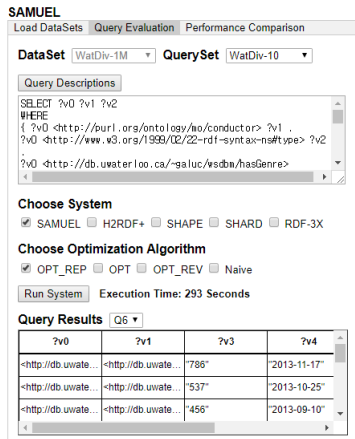


Figure 4: SAMUEL GUI

assigned to each reducer and the cost of each RDF join operation, implemented with binary hash join, is computed by summing the sizes of two input RDF triple pattern lists. Triples are grouped on the basis of distinct triple patterns and the groups are ordered in a descending order of cardinality. We devise a heuristic optimization algorithm based on *first-fit decreasing* scheme whose input is a set of ordered lists of the distinct triple pattern groups. Some triple patterns have very high cardinality as shown in Fig 3. We thus allow splitting a very large triple pattern list into multiple small lists for better workload balance. Note that our global query plan tree is gradually built. That is, the n -th level in the tree is computed right before running the n -th joining phase. In each joining phase, implemented by another M/R job, mappers read grouped RDF triples and tag reducer IDs to the triples according to the global plan tree. Since mapped outputs are shuffled by intermediate keys, triples tagged by the same reducer ID go to the same reducer together and are then joined. Again, each reducer computes the cardinality of its joined outputs and the cardinality information and joined results are stored in HDFS. Then, our optimizer repeats building the $n+1$ -th level of the plan tree with the cardinality of the results of the n -th joining phase.

4 DEMONSTRATION

During the demonstration, audience are invited to compare our system with other distributed SPARQL engines based on the MapReduce framework, interacting with the system to run queries and to check the influences of optimization techniques.

Hardware setup: We implemented our system with Hadoop version 1.2.1. SAMUEL as well as compared systems were installed and run on a cluster of 15 nodes, each of which was equipped with an Xeon E5-2620 2.1GHz CPU, 64GB memory and an 1TB 7200RPM HDD, running on Ubuntu 12.04. All the nodes were connected via Gigabit switching hub and a node is designated as a master node. We basically used the same settings for our cluster for fair comparison. However, the settings were sometimes tuned for showing the best performance of compared systems.

Dataset: We used two datasets, which had been widely used for measuring SPARQL engines in the literature [4, 10]. Table 1 and Figure 3 present the statistics for datasets used in our demonstration and their data distributions. Both datasets exhibited high skewness in their data distributions in that only a few of triple patterns dominated most of the data distributions (see Fig. 3).

Compared systems: In our demonstration, we compare our system with RDF-3X [17], a single-machine SPARQL engine that utilizes various indexes, and three other MapReduce-based systems, *i.e.*, SHARD [21], SHAPE [14], and H2RDF+ [19]. For evaluation, the performance of each system was averaged over five runs excluding the maximum and minimum values.

Demo scenarios and interaction

We provide a user interface shown in Fig. 4 to demonstrate the performance of SAMUEL using large-scale RDF datasets, *i.e.*, LUBM and WatDiv. In our demonstration, we however use only a few fractions of the two datasets due to the limited time and computing resources. However, we still present our evaluation results performed with all the datasets (see Fig 5). Currently, SAMUEL supports a subset of SPARQL language, *i.e.*, basic graph pattern matching. In our demonstration, users will be given a list of SPARQL queries generated from WatDiv for the datasets in Table 1. Users are also allowed to load their own queries and RDF data into the system and run the queries themselves. During the processing, users will be explained with Hadoop GUI and our own UI how our system processes multiple SPARQL queries simultaneously. Users will also check how features of SAMUEL affect the overall performance as they turn on and off the features, *i.e.*, sharing input scan and filtered solutions, optimization policies, and so on.

REFERENCES

- [1] 2008. SPARQL Query Language for RDF. <https://www.w3.org/TR/rdf-sparql-query/>. (2008).
- [2] 2014. Resource Description Framework (RDF). <https://www.w3.org/RDF/>. (2014).
- [3] Ibrahim Abdelaziz, Razen Harbi, Zuhair Khayyat, and Panos Kalnis. 2017. A Survey and Experimental Comparison of Distributed SPARQL Engines for

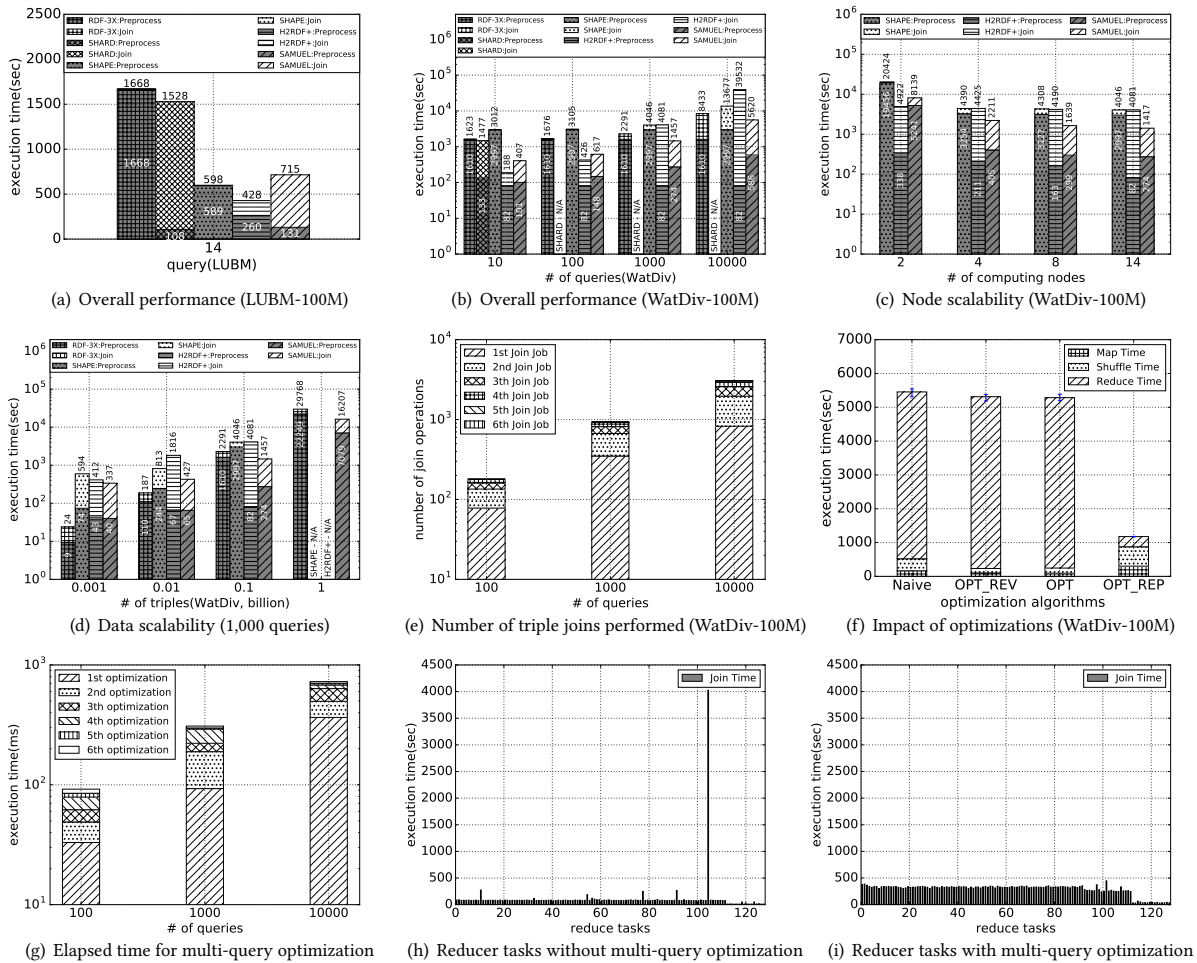


Figure 5: Performance evaluation

- Very Large RDF Data. *Proceedings of the VLDB Endowment* 10, 13 (2017).
- [4] Güneş Aluç, Olaf Hartig, M Tamer Özsu, and Khuzaima Daudjee. 2014. Diversified stress testing of RDF data management systems. In *International Semantic Web Conference*. Springer, 197–212.
- [5] David Applegate and William Cook. 1991. A computational study of the job-shop scheduling problem. *ORSA Journal on computing* 3, 2 (1991), 149–156.
- [6] François Belleau, Marc-Alexandre Nolin, Nicole Tourigny, Philippe Rigault, and Jean Morissette. 2008. Bio2RDF: towards a mashup to build bioinformatics knowledge systems. *Journal of biomedical informatics* 41, 5 (2008), 706–716.
- [7] Emmanuel Boutet, Damien Lieberherr, Michael Tognolli, Michel Schneider, and Amos Bairoch. 2007. UniProtKB/Swiss-Prot: the manually annotated section of the UniProt KnowledgeBase. *Plant bioinformatics: methods and protocols* (2007), 89–112.
- [8] Hyebyong Choi, Kyong-Ha Lee, Soo-Hyong Kim, Yoon-Joon Lee, and Bongki Moon. 2012. HadoopXML: a suite for parallel processing of massive XML data with multiple twig pattern queries. In *Proceedings of the 21st ACM international conference on Information and knowledge management*. ACM, 2737–2739.
- [9] François Goasdoué, Zoi Kaoudi, Ioana Manolescu, Jorge-Arnulfo Quiané-Ruiz, and Stamatis Zampetakis. 2015. Cliquesquare: Flat plans for massively parallel RDF queries. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*. IEEE, 771–782.
- [10] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. 2005. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web* 3, 2 (2005), 158–182.
- [11] Mohammad Husain, James McGlothlin, Mohammad M Masud, Latifur Khan, and Bhavani M Thuraisingham. 2011. Heuristics-based query processing for large RDF graphs using cloud computing. *IEEE Transactions on Knowledge and Data Engineering* 23, 9 (2011), 1312–1327.
- [12] Yongchul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. 2012. Skewtune: mitigating skew in mapreduce applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 25–36.
- [13] Wangchao Le, Anastasios Kementsitsidis, Songyun Duan, and Feifei Li. 2012. Scalable multi-query optimization for SPARQL. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*. IEEE, 666–677.
- [14] Kisung Lee and Ling Liu. 2013. Scaling queries over big RDF graphs with semantic hash partitioning. *Proceedings of the VLDB Endowment* 6, 14 (2013), 1894–1905.
- [15] Kyong-Ha Lee, Yoon-Joon Lee, Hyunsik Choi, Yon Dohn Chung, and Bongki Moon. 2012. Parallel data processing with MapReduce: a survey. *ACM SIGMOD Record* 40, 4 (2012), 11–20.
- [16] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick Van Kleef, Sören Auer, et al. 2015. DBpedia—a large-scale, multilingual knowledge base extracted from Wikipedia. *Semantic Web* 6, 2 (2015), 167–195.
- [17] Thomas Neumann and Gerhard Weikum. 2010. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal?The International Journal on Very Large Data Bases* 19, 1 (2010), 91–113.
- [18] Tomasz Nykiel, Michalis Potamias, Chaitanya Mishra, George Kollios, and Nick Koudas. 2010. MRShare: sharing across multiple queries in MapReduce. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 494–505.
- [19] Nikolaos Papiailiou, Ioannis Konstantinou, Dimitrios Tsooumakos, Panagiotis Karras, and Nectarios Koziris. 2013. H2RDF+: High-performance distributed joins over large-scale RDF graphs. In *Big Data, 2013 IEEE International Conference on*. IEEE, 255–263.
- [20] Xuguang Ren and Junhu Wang. 2016. Multi-query optimization for subgraph isomorphism search. *Proceedings of the VLDB Endowment* 10, 3 (2016), 121–132.
- [21] Kurt Rohloff and Richard E Schantz. 2010. High-performance, massively scalable distributed systems using the MapReduce software framework: the SHARD triple-store. In *Programming Support Innovations for Emerging Distributed Applications*. ACM, 4.
- [22] Alexander Schätzle, Martin Przyjacieli-Zablocki, Simon Skilevic, and Georg Lausen. 2016. S2RDF: RDF querying with SPARQL on spark. *Proceedings of the VLDB Endowment* 9, 10 (2016), 804–815.
- [23] Guoping Wang and Chee-Yong Chan. 2013. Multi-query optimization in mapreduce framework. *Proceedings of the VLDB Endowment* 7, 3 (2013), 145–156.

GEDetector: Early Detection of Gathering Events Based on Cluster Containment Join in Trajectory Streams

Bin Zhao
School of Computer Science and Technology
Nanjing Normal University, China
zhaobin@njnu.edu.cn

Genlin Ji*
School of Computer Science and Technology
Nanjing Normal University, China
glji@njnu.edu.cn

Yu Yang
School of Computer Science and Technology
Nanjing Normal University, China
162202020@njnu.edu.cn

Zhaoyuan Yu
Key Laboratory of Virtual Geographic Environment Ministry of Education
Nanjing Normal University, China
yuzhaoyuan@njnu.edu.cn

Xintao Liu
Department of Land Surveying and Geo-Informatics
The Hong Kong Polytechnic University, Hong Kong
xintao.liu@polyu.edu.hk

Ningfang Mi
Electrical and Computer Engineering Department
Northeastern University, Boston, MA USA
ningfang@ece.neu.edu

ABSTRACT

Existing trajectory patterns, such as flock, convoy, swarm, and gathering, are to detect moving clusters staying or travelling together for a certain time period. But these patterns model group movement behaviors after moving objects' gathering together. This may result in losing golden opportunities to detect emergency incidents earlier, such as traffic congestion and serious stampedes. In this work, we propose a novel group pattern, called converging, which can model converging behaviors of moving objects. As a proof-of-concept, we implemented a visual analytic system GEDetector based on trajectory streams to detect gathering events as early as possible. A user-friendly interface is designed to help users gain insights into gathering events from spatial and temporal aspects. Finally, we demonstrate the effectiveness and efficiency of our system by using a real world dataset.

1 INTRODUCTION

The existing group pattern mining methods, such as flock [1], convoy [2], swarm [3], travelling companion [4], and gathering [5], are to discover a group of objects that stay or travel together for a certain time period [6]. Analysis tasks based on group patterns can be used for enormous applications, including transportation optimization, public security, advertisement delivery, travel recommendation, and animal movement study and so on [7]. In this work, we develop a visual analytic system GEDetector to detect gathering events in trajectory streams.

Previous studies on group patterns are unsuitable for detecting gathering events despite of wide applications of group patterns. It is worth pointing out that, most of group patterns can capture group movement behaviors only after moving objects gather together. For example, convoys require that each group of objects travel together during the whole pattern lifetime. Gatherings also require that all participators stay together at the most timestamps in the pattern lifetime. However, in real life, people are more interesting in modelling and identifying group converging behaviors before the gathering events actually happen, since

this helps to proactively report and handle the coming public incidents, such as traffic congestion and serious stampedes.

However, the efficient discovery of convergings in trajectory streams is a challenging task due to two main reasons: (1) the existing group patterns cannot capture intuitively converging behaviors of gathering events; (2) the discovery of convergings may face huge search space and incur high cost. In this work, we propose a novel group pattern, called converging, which can model the converging behaviors of moving objects and detect gathering events in trajectory streams. Furthermore, we propose a converging pattern mining method based on cluster containment join (CCJ), which utilizes a signature quad-tree based index, called SQTI, to organize clusters hierarchically and spatially. The SQTI based mining algorithm enables us to rapidly reduce search space and efficiently identify interesting and useful converging patterns.

In this demo, we present a novel Gathering Event Detection system, named GEDetector, which is designed to achieve an early detection of gathering events through mining converging patterns in trajectory streams. The GEDetector has been deployed on a virtual machine of Alibaba Cloud, which can be accessed at <http://103.242.175.191:5456/gedetector>.

2 PROBLEM FORMULATION

The identifying characteristic of a gathering event is a *converging*, which is a group of moving objects gathering from different directions for at least k_t timestamps. It is easy to see that, converging patterns focus on the earlier stages of gathering events compared to other existing patterns. To accurately model converging behaviors, we introduce *participators*, which are used to indicate the objects appearing in at least k_p consecutive clusters of this converging, and require that a converging should contain at least k_m participators.

Now we use Fig. 1 as an example to illustrate the converging pattern, and let $k_t = 2$, $k_p = 2$, $k_m = 4$, in which there are six moving objects joining a gathering event from different directions and forming one cluster. Additionally, there are two clusters (i.e., c_1 & c_2) that are gathered at t_2 and later join the cluster c_3 at t_3 . Such set containment between two clusters is called *cluster containment relation*, denoted by \subseteq_c . Thus, we have $c_1 \subseteq_c c_3$ & $c_2 \subseteq_c c_3$ in Fig. 1(a). By joining these two cluster containment relations, we can construct a containment tree to model converging behaviors of moving objects, as shown in Fig.

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

1(b). To realistically model converging behaviors, a converging requires to have enough participators who appear in the most snapshot clusters during the event lifetime. As shown in Fig. 1(c), the group of participators $\langle o_1, o_2, o_3, o_4, o_5 \rangle$ satisfies the above requirements because they appear in two clusters (i.e., c_1 & c_2) at time t_2 and t_3 . But, the group excludes o_6 , since o_6 only appears in one snapshot cluster, which is less than the given threshold $k_p = 2$.

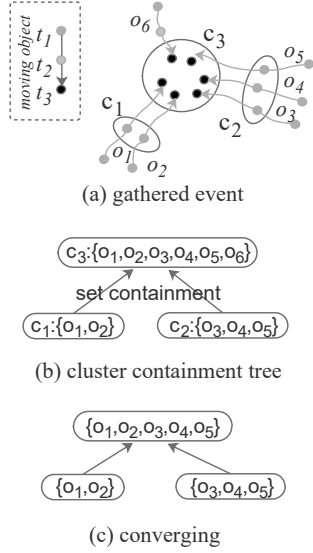


Figure 1: Example of a converging

3 FRAMEWORK OF GEDETECTOR

Fig. 2 shows the framework of GEDetector, which consists of three major modules: (1) **snapshot cluster discovery (SCD)**: we perform density-based clustering on the moving objects to discover a set of snapshot clusters at each timestamp; (2) **cluster containment join (CCJ)**: we perform a cluster containment join on any two snapshot cluster sets at consecutive timestamps, and return a collection of cluster containment relations; and (3) **converging detection (CD)**: we construct cluster containment trees through joining all cluster containment relations, and further derive the qualifying results from the candidates of cluster containment trees according to the aforementioned requirements of converging concepts.

The detection procedure of GEDetector is as follows. The initial input of GEDetector is a trajectory data stream that is periodically collected at timestamp t_i . When the latest batch of trajectory data is appended to the trajectory database, GEDetector will trigger the detection process in an incremental manner, as shown in Fig. 2. At t_i , the snapshot clusters are first discovered from the new trajectory data by clustering algorithms. Then, the CCJ is performed between the snapshot cluster sets of t_{i-1} and t_i . Correspondingly, the new convergings may be generated, and some existing convergings may be also updated. The procedures of the first two modules (i.e., SCD & CCJ) can be repeated until no new trajectory data is given. Finally, the qualified converging patterns are derived by the third module (CD).

The efficient discovery of convergings in trajectory streams is a challenging task. First, it is hard to discover all moving objects

attending a gathering event in the same way as many state-of-the-art trajectory clustering methods since they do not stay close together most of the time. Second, to achieve early detection, converging algorithms need to detect gathering events in an incremental manner. Third, identifying cluster containment relations between any consecutive timestamps may face huge search space and hence incur high cost. Fourth, the system has to ensure the effectiveness of mining converging patterns from trajectory streams.

To tackle these issues, the GEDetector employs a general framework for effective and efficient discovery of convergings. Specially, To boost the performance of CCJ, which is a fundamental part of mining converging patterns, we develop a signature quad-tree based index, called SQT, to organize clusters hierarchically and spatially, and correspondingly propose an SQT based CCJ (SQTCCJ) approximate algorithm, which enables us to rapidly filter unqualified candidates and efficiently identify matches by considering cluster containment relationship and spatial proximity simultaneously. In addition, to facilitate evaluating set containment, we approximate the sets of moving objects by means of a signature technique, and use a bloom filter method [8] to generate signatures in this work.

The SQTCCJ consists of two phases: (1) SQT construction; and (2) SQT probing and verification. Firstly, we construct an SQT structure to organize all candidate clusters. Then, SQTCCJ performs a cluster query on SQT for each query cluster to obtain all cluster containment matches.

SQT Construction. An SQT is constructed in the following manner. We start with a set of candidate clusters, and insert representative points of these clusters into a spatial region according to their positions. Then we partition the whole spatial region by recursively subdividing it into four square quadrant cells (i.e., NE, NW, SE, and SW), until the number of points in it meets a certain threshold, ρ . Otherwise, the quadrant cell continues splitting into four small ones. The tree structure of SQT follows the above spatial decomposition. Although the SQT is organized in the same way as the traditional quad-tree, the difference is that each node is assigned a signature consisting of m bits, which represents a set of moving objects of all clusters in the corresponding cell. In the next phase, we will utilize signature comparisons to support membership query.

EXAMPLE 1. We use the data in Fig. 3(a)(b) to illustrate the construction of SQT. Based on these clusters, we build an SQT for searching clusters, and its nodes are organized as Fig. 3(c)(d) shows.

SQT Probing and Verification. The search process based on SQT works in a top-down manner instead of in a recursive manner. And its goal is to find the nearest quadrant cell to the query point. Specifically, when a query cluster q comes, the search starts at the root node and utilize the query signature and its coordinates to probe the index structure. When it visits a internal node p with signature $p.sig$, we need to check it to see if $q.sig \vee p.sig = p.sig$. If yes, it immediately performs a four-way comparison operation at the node, and then chooses the subtree where the centroid of its corresponding MBR is nearest to the query point. Otherwise, it visits the sibling nodes. When it reaches a leaf node, we need to verify all clusters to check if there exists a super-cluster in the leaf node. If yes, we can obtain a cluster containment match; otherwise, we get a mismatch.

EXAMPLE 2. Continuing with Example 1, we illustrate how to perform the query process based on SQT. For a given query

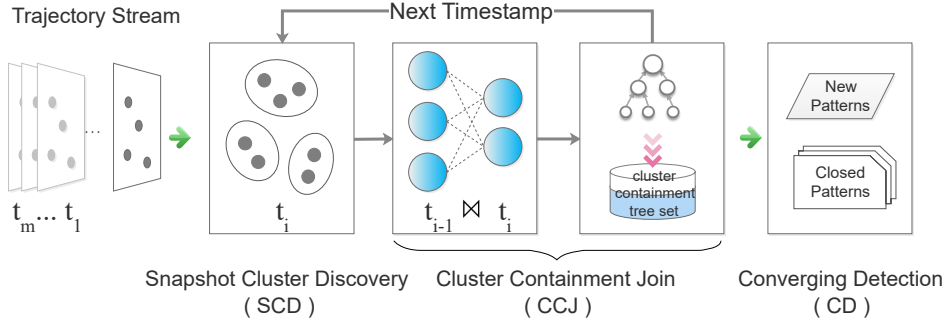


Figure 2: Framework of GEDetector

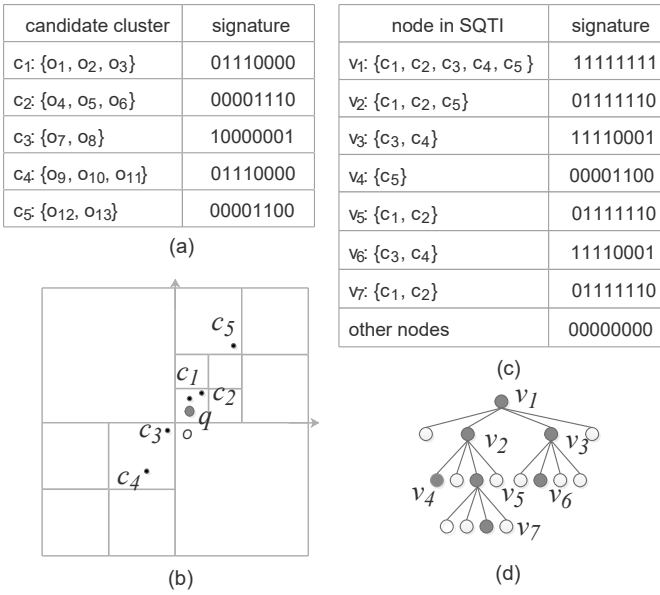


Figure 3: Illustration of SQTI construction

$q = \{o_1, o_2\}$ with the signature 01110000, the search starts at the root node v_1 . Then it need to check the nodes v_2 and v_3 to see if the query signature is contained. Unfortunately, both v_2 and v_3 satisfy the above condition. In this case, the search algorithm in the traditional tree will suffer from a backtracking problem incurred by a recursive paradigm. But, the search in *SQTCJ* can avoid this since it only takes the nearest quadrant cell to the query position. As a consequence, the search chooses the sub-tree rooted at the node v_2 . It reaches the leaf node v_7 along the path $v_1 v_2 v_5 v_7$ after signature comparisons. At the leaf node, we still need to verify all clusters in it because of false positive. Finally, we can obtain a match $q \subseteq c_1$.

4 DEMONSTRATION

4.1 Demo Interface

In the demo of GEDetector, we provide a monitoring map interface for users to visualize and monitor gathering events dynamically. A screenshot of the demo is shown in Fig. 4. This demo interface allows users to investigate gathering events from various views, including spatial view, temporal view, and detail view. The map on the upper left pane of the GUI visualizes the discovered gathering events using map markers, which represent the location points of all gathering events. The bar chart on the bottom left pane of the GUI represents the basic characteristics

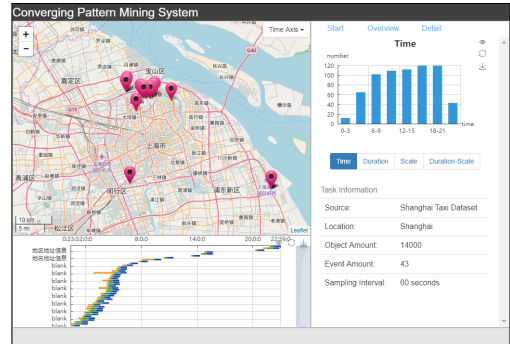


Figure 4: Screenshot of main interface

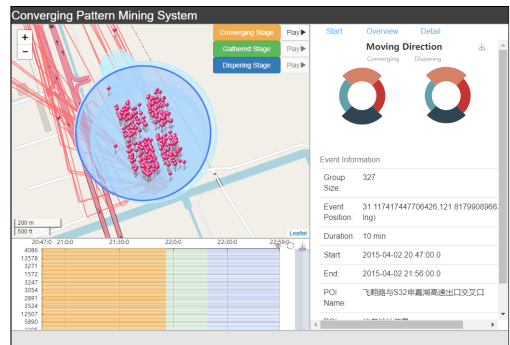


Figure 5: Visualization of a taxi gathering event at Shanghai Pudong international airport

of gathering events, such as scale, start time, durability, and point of interest (POI) type. When the user clicks on any map marker, the interface switches to the display mode of the corresponding gathering event. The user can obtain more detailed information of the gathering event, including the routes of all participators, direction statistics of all routes (represented by a wind rose diagram), the event timeline, and other properties. As shown in Fig. 5 and 6, our system has detected a typical gathering event at Shanghai Pudong international airport, where a large number of taxis converge together for waiting guests.

In summary, the GEDetection system can allow users to interactively monitor and visualize gathering behaviours in various ways, and also help users gain insights of the identified gathering events from both spatial and temporal aspects.

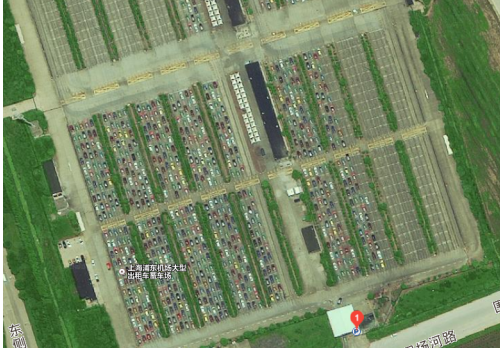


Figure 6: Satpic of a cabstand at Shanghai Pudong international airport

5 PERFORMANCE OF GEDETECTOR

To evaluate the performance of our system, we conduct all experiments based on a real trajectory dataset, namely Shanghai taxi cab traces (BigTaxi), which is sourced from the one generated by 13,000 taxis of Shanghai in a period of 30 days (from April 1st to April 30th in 2015). In this section, we investigate the efficiency of the SQTCCJ algorithm, which is the essential part of our system.

The SQTCCJ algorithm may generate approximate results due to missing some cluster containment matches, only when a result cluster does not share the same cell with the query. Therefore, we use an accuracy rate metric, which is the fraction of matches obtained by SQTCCJ over the total amount of those of NLCCJ, to evaluate the result of SQTCCJ algorithm. In Fig. 7(a), we show accuracy rates of SQTCCJ. We can find that SQTCCJ can find almost all candidate clusters when the parameter ρ is greater than 20.

Next, in Fig. 7(b)(c), we show another nice property of SQTCCJ, namely insensitivity to key parameters including the signature length m and the number threshold ρ . In addition, we note that we can tackle the problem of CCJ using set containment join methods if we treat a cluster as a set. Thus, we compare our algorithm with TT-Join [9], which is the state-of-the-art method evaluating set containment join. As we can see from Fig. 7(d), SQTCCJ significantly outperforms TT-Join especially as the dataset size increases.

6 ACKNOWLEDGEMENTS

This study was supported by the National Natural Science Foundation of China under grant numbers 41471371, 41571379, and the PAPD program.

REFERENCES

- [1] Patrick Laube and Stephan Imfeld. Analyzing relative motion within groups of trackable moving point objects. In *Geographic Information Science, Second International Conference, GIScience 2002, Boulder, CO, USA, September 25-28, 2002, Proceedings*, pages 132–144, 2002.
- [2] Hoyoung Jeung, Man Lung Yiu, Xiaofang Zhou, Christian S. Jensen, and Heng Tao Shen. Discovery of Convoys in Trajectory Databases. *Proc. VLDB Endow.*, 1(1):1068–1080, August 2008.
- [3] Zhenhui Li, Bolin Ding, Jiawei Han, and Roland Kays. Swarm: Mining Relaxed Temporal Moving Object Clusters. *Proc. VLDB Endow.*, 3(1-2):723–734, September 2010.
- [4] L. A. Tang, Y. Zheng, J. Yuan, J. Han, A. Leung, C. C. Hung, and W. C. Peng. On Discovery of Traveling Companions from Streaming Trajectories. In *2012 IEEE 28th International Conference on Data Engineering (ICDE)*, pages 186–197, April 2012.
- [5] K. Zheng, Y. Zheng, N. J. Yuan, and S. Shang. On discovery of gathering patterns from trajectories. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 242–253, April 2013.

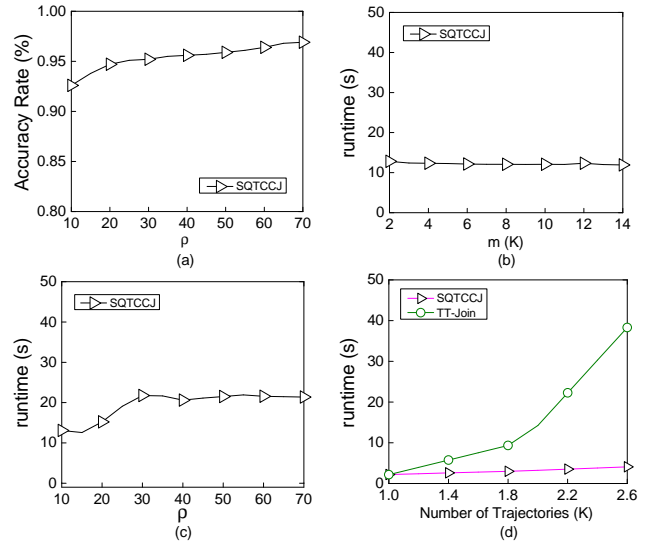


Figure 7: Performance of SQTCCJ

- [6] Qi Fan, Dongxiang Zhang, Huayu Wu, and Kian-Lee Tan. A General and Parallel Platform for Mining Co-movement Patterns over Large-scale Trajectories. *Proc. VLDB Endow.*, 10(4):313–324, November 2016.
- [7] Yu Zheng. Trajectory Data Mining: An Overview. *ACM Transactions on Intelligent Systems and Technology*, 6(3):1–41, May 2015.
- [8] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.
- [9] Jianye Yang, Wenjie Zhang, Shiyu Yang, Ying Zhang, and Xuemin Lin. Tt-join: Efficient set containment join. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*, pages 509–520, 2017.

Reconciling Privacy and Data Sharing in a Smart and Connected Surrounding

Paul Tran-Van^{1,2,3}
¹Cozy Cloud, France
 paul@cozyccloud.cc

Nicolas Anciaux^{2,3}
²Inria, France
 nicolas.anciaux@inria.fr

Philippe Pucheral^{2,3}
³U. Versailles St-Q., France
 philippe.pucheral@uvsq.fr

1 INTRODUCTION

When Alice goes trekking in the French Alps with friends, she is equipped with a pedometer to measure her efforts, takes pictures using her smartphone and uses a mobile coach app to monitor her trip and GPS trail. But how could Alice have a transversal view of the personal data she generates and how could she share -part of- her data with her friends?

The Personal Cloud paradigm emerges[1] (e.g., Cozy Cloud, ownCloud, Databox to cite a few) and holds the promise of a Privacy-by-Design storage and computing platform where each individual could gather her complete digital environment in one place and share it under control. Conjointly, smart disclosure initiatives pushed by legislators (e.g., EU General Data Protection Regulation) and industry-led consortiums (e.g., Blue Button for medical records in the US, Midata in the UK, MesInfos in France) give shape to this paradigm by letting individuals getting their personal data back from the applications that collected them. Hence, Alice could link her personal devices to the personal cloud platform of her choice and then manage the personal data she generates when trekking and regulate data sharing at will. However, the personal cloud paradigm causes a gravity shift of data management and data security from organizations to individuals, who are usually not database administrators nor security experts. Unfortunately, the main existing access control models (e.g., RBAC, ABAC or TBAC [2]) are geared towards central authorities and require a deep expertise to define users, roles and privileges. Some decentralized models have been proposed to let individuals manually define their own sharing preferences, often based on Web of Trust-like approaches [8] or on the owner's social graph [3], but offer limited expressive power and poorly cope with the versatile nature of the personal cloud. To tackle this issue, several works aim to ease the sharing administration. For example, [5, 12] give the possibility for the owner to share any kind of personal data through the use of attribute-based sharing rules, while [4, 7] explore machine learning techniques to automatically infer the best sharing policies. However, they provide little means for individuals to control the actual effects of their policies and could actually result in unexpected data leakage. This contradicts a founding principle of the Personal Cloud paradigm, namely enabling individuals making sovereign decisions about the sharing of their data [1]. The problem is exacerbated in a ubiquitous and smart surrounding producing continuous flow of daily activity events.

We derive from these statements a new sharing paradigm dedicated to the personal cloud context, called SWYSWYK (Share What You See with Who You Know). SWYSWYK relies on two founding principles: (1) provide intuitive means to derive sharing rules directly from the personal cloud content and help the personal cloud owner administer the resulting sharing policy by

visualizing and sanitizing its net effects and (2) provide a secure personal cloud architecture giving tangible guarantees that the sharing policy will be properly enforced, whatever the security expertise of the owner.

In [11], we investigated point (2) and proposed a secure architecture combining untrusted, isolated and secure execution environments. [10] presented a practical instantiation of this architecture where the reference monitor runs into a secure hardware device. In [9], we focus on point (1) and introduce the semantics of the SWYSWYK sharing paradigm and discuss the specificities of its administration.

This demonstration focuses on point (1) with the goal to assess the practical interest of our paradigm. To this end, we have integrated SWYSWYK in a real personal cloud platform, namely Cozy, and apply it to a smart surrounding scenario inspired by Alice's one. A video of the demonstration is available online¹. We refer to [10] for a demonstration focused on the security and scalability of the platform in a constrained environment.

In this paper, Section 2 presents the SWYSWYK baseline, Section 3 gives the scenario and Section 4 concludes.

2 SWYSWYK PARADIGM

2.1 Baseline

SWYSWYK is not yet-another access control model. It is rather a new sharing paradigm, helping the derivation of expressive access control rules directly from the personal cloud content and providing convenient tools to administrate the resulting policies. The originality of SWYSWYK relies on two core principles helping circumventing the aforementioned difficulties of data sharing in the Personal Cloud context:

Documents are rules. The personal cloud content on its own conveys intuitive sharing rules, e.g., share pictures and related events of a trek with people who appear on these pictures and as such are identified as participants in that trek. Such rules should be straightforward to express, as the related permissions could be derived from the documents' content. The subjects targeted by the document, called identifyees [6], should be extracted from the document content and enter in the rule definition. We call *reflexive sharing rules* the rules based on this principle.

Subjects and objects are documents. The content of a personal cloud also intrinsically describes the individual's acquaintances under different forms (e.g., contact files, identity picture.) and conversely, acquaintances are associated with pieces of information in the owner's space (e.g., agenda entries, photos on which a friend appears). A corollary is that for each permission granted to a subject *s* on an object *o*, viewable documents should represent *s* and *o*. More generally, the result of a sharing policy (sets of sharing rules) must be viewable by the personal cloud owner, who can thus precisely understand what is the net effect of this policy. For example, a stream of GPS tracks may be represented

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

¹http://wanda.inria.fr/demos/videos/swyswyk_model.org

as trajectories on a map and time series of activities logs could be represented in graphs.

The combination of these principles gives substance to the *Share What You See with Who You Know* (SWYSWYK) paradigm. This is in direct opposition with existing approaches that are either very limited and manual, or based on complex and opaque computations that cannot be easily understood by the end user.

2.2 Sharing Paradigm Semantics

SWYSWYK aims at providing simple expressions for sharing rules and make the sharing policy self-administrated when the personal cloud content evolves. We show here how this can be captured within simple semantics, combined with a set of simplifying assumptions. First, our sharing paradigm relies on a closed policy, i.e. every action not explicitly granted is denied. Actions are CRUD operations on documents in the personal cloud. The paradigm supports only authorizations (positive rules) but allows the owner to post-filter the produced Access Control List (ACL) when exceptions need to be declared. Consequently, there is a direct translation between sharing rules and sets of ACLs: an action a is granted to subject s on document d iff $(s, d, a) \in ACL$ and is denied otherwise. The sharing is by construction consistent (the decision is unique), complete (the decision always exists) and can be evaluated in logarithmic time.

For the sake of conciseness, we do not formally define here all the notations and operators of SWYSWYK. Rather, we illustrate the paradigm through a single type of sharing rules, namely the reflexive sharing rules, and refer to [9] for a complete description.

Reflexive sharing rules. These rules express the sharing of documents with subjects appearing on it. They implement the documents are rules principle and are thus considered as first-class citizen rules:

$$ACL \leftarrow \{(s, d, a) \in S \times D \times A / Filter(d, Q) \wedge MatchS(DI(d), SI(s))\}$$

$Filter$ and DI are user-defined, platform dependent, functions. $Filter$ returns true if a document d of the personal cloud satisfies the qualification Q , that can be expressed on the metadata or on the content of d . DI extracts identification traits of individuals, denoted next by IT , from d . IT must uniquely represent a subject in the personal cloud and can combine simple attributes (e.g., email, phone number) or complex representation (e.g., facial features, fingerprint). SI and $MatchS$ are internal SWYSWYK operators. SI returns the IT from a registered subject s and $MatchS$ returns true if the compared IT s are equivalent. Below are various illustrations of reflexive sharing rules.

Example 1. Share the pictures taken during my trekking sessions with the people appearing on it:

$Q: docType='photo' \wedge tagGallery='trek'$

$DI: face\ detection\ algorithm$

$MatchS$: here, compares the facial features extracted from DI with the ones returned by SI from known subjects.

Example 2. Share the minutes of meetings with the attendees:

$Q: docName\ like\ 'minutes-\%.doc'$

$DI: extract\ attendee\ names\ from\ a\ minute\ document$

These two examples, extracted from two different application contexts, show the generality of the sharing paradigm.

2.3 Sharing Administration

To make the sharing paradigm practical, subject declaration and maintenance should be (quasi) automatic while respecting the owner's privacy. In SWYSWYK, the notion of rule consistency

concretizes the fact that the effects of all rules can be visualized (and then easily controlled by the owner), the notion of exceptions permits customization of these effects according to the owner's preferences, and subject administration can be automatically performed such that the set of subjects grows along document insertions and rule declarations with minimal interactions.

Rules Consistency. A SWYSWYK sharing rule is said well-formed iff it only produces ACLs involving viewable documents shared with recognizable subjects: $\forall sr \in SR, \forall acl \in ACL, acl.d \in DV \wedge acl.s \in DS$, where SR is the set of sharing rules, DV the set of viewable documents and DS the subset of viewable documents characterizing a unique subject. Any acl which does not satisfy this condition is filtered out.

Rules Exceptions. Instead of introducing interdiction rules to capture exceptions, which makes the net effects of the resulting policy complex to apprehend, we simply give the owner the ability to filter out the permissions which hurt her privacy (considered as suspicious ACLs). We introduce three types of watchdog triggers to highlight suspicious permissions:

$$What(Q_s, A) \rightarrow \{(s, \{(d, a)\}) / (s, d, a) \in ACL^* \wedge s \in Q_s(S) \wedge a = A\}$$

$$Who(Q_d, A) \rightarrow \{(d, \{(s, a)\}) / (s, d, a) \in ACL^* \wedge d \in Q_d(D) \wedge a = A\}$$

$$Which(Q_s, Q_d, A) \rightarrow \{(s, d, a) / (s, d, a) \in ACL^* \wedge s \in Q_s(S) \wedge d \in Q_d(D) \wedge a = A\}$$

ACL^* corresponds to the set of newly created/updated ACLs. $What$ identifies, for each sensitive subject, the new set of (document, action) she is granted to (e.g., *which new documents can be seen by my boss?*). Who identifies, for each sensitive document, the new set of subjects s with granted action a on them (e.g., *which new subjects have a read access to my medical records?*). Finally, $Which$ identifies new ACLs combining a selection of (sensitive) subjects and documents (e.g., *which new authorizations my colleagues have on my family photos?*).

Subjects Administration. New subjects can automatically be created while inserting new contact files or address book entries. The IsS operator is invoked each time (1) documents are created or updated in the personal cloud and (2) a new rule invoking IsS is defined, thus enriching the set of subject S along document insertions and rule declarations as side-effects of the function. Each $s \in S$ is made of the extracted identification traits and at least one generated credential for the authentication.

2.4 Sharing Enforcement

General principle. The creation, maintenance and evaluation of a set of SWYSWYK permissions are as follows: (1) the owner creates sharing rules and watchdog triggers to be applied on her personal cloud; (2) a *rule translator* translates the selected rules into candidate (ACL^*) and suspicious ($ACL^?$) ACLs and materialize them; (3) the owner checks the suspicious ACLs at will and accepts (ACL^+) or rejects (ACL^-) them using the *administration GUI*; (4) the *reference monitor* authenticates subjects and evaluates *Allowed* (i.e., $Allowed(s, d, a) = true$ iff $(s, d, a) \in ACL^+$) and delivers the requested documents accordingly.

ACL production and maintenance. Five operators are required to translate any sharing rule into ACLs, namely $Filter$, DI , SI , IsS and $MatchS$. The data flow between the operators to translate a reflexive sharing rule into ACLs is shown in Figure 1. At declaration time, the rule tree is evaluated over all documents of the personal cloud. First, $Filter$ operators are evaluated at the leaf of each branch to select the targeted subjects documents (right

branch) and the targeted objects documents containing identifiers (left branch). Then *DI* operators extract the list of *ITs* from the targeted subject documents (left branch) and from the objects documents (*ITs* of the identifiers). In the right branch, *IsS* tries to match the extracted *ITs* with the subjects already registered in *S*, then *SI* appends the identified *ITs* to each subject. Finally, *MatchS* joins the left and right branches on subject *ITs* and produces the (candidate) ACLs. At insertion of a new document *d* in the personal cloud, the *Filters* of all rules are evaluated against *d* to check whether new candidate ACLs can be produced.

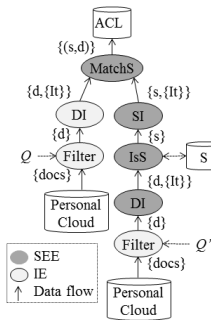


Figure 1: ACL production

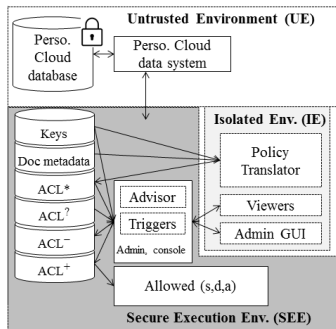


Figure 2: Reference architecture

Secure enforcement of sharing policies. The enforcement issue is exacerbated when the personal cloud platform runs on the owner’s side, the security of which can be questioned. In [10], we proposed a reference architecture tackling this issue, displayed in Figure 2. It consists of three environments: (i) an *untrusted environment (UE)* on which no security assumption is made for the code nor for the data, (ii) an *isolated environment (IE)* on which general purpose code can be run with the guarantee that it cannot leak any information but with no guarantee about the soundness and honesty of its output and (iii) a *Secure Execution Environment (SEE)* which runs only certified core programs and protects data and code against snooping and tampering. In Figure 1, the light grey operators (*DI* and *Filter*) are run in the *IE*, as they are made of untrusted third parties code, while the ones in dark grey are executed in the *SEE*. Administrative tools helping the owner to control and sanitize the ACLs are partly hosted in *SEE* (e.g., watchdog triggers) and in *IE* (e.g., document viewers).

3 DEMONSTRATION

The objective of this demonstration is to show that the SWYSWYK paradigm makes sense in concrete environments. Hence, it considers a ubiquitous surrounding scenario with connected smart devices producing continuous data streams that are gathered and stored in the Cozy personal cloud platform. SWYSWYK has been partially integrated in Cozy, a simplified version being part of the server stack².

3.1 Demonstration platform

The platform consists of an Android smartphone, a Withings smart watch and a local Cozy instance running on a laptop with Ubuntu 16.04. Additionally, several Cozy instances running on a remote server are used to simulate other subject’ personal clouds. Pictures and GPS tracks are synced with the local Cozy instance thanks to the Android Cozy app. Pedometer data from the smartwatch is retrieved through the Withings’ API. The Cozy stack running on the laptop is implemented in Go and stores documents in a CouchDB database. The Cozy apps are developed in JavaScript, with the React framework.

3.2 Demonstration scenario

The demonstration focuses on the usage of the SWYSWYK paradigm. The scenario is composed of four steps, as summarized in Figure 3 and described in the video accessible online:

Step 1 - Data collection: this step illustrates how surrounding data produced by smart devices can be collected by the Cozy platform to be further exploited. A Cozy instance is populated with a set of predefined documents and timely integrates data produced by Paul’s smartphone and pedometer. A Sharowalky application developed on Cozy (for illustration purpose) manages Paul’s trekking data, namely his photos, GPS trails and physical activity. The attendees are invited to connect to Cozy as Paul (the personal cloud owner and incidentally co-author of this paper), open the Sharowalky app and browse days of trekking.

Step 2 - Sharing definition: the Sharowalky app proposes the attendee to share the photos of a trekking day with Paul’s friends appearing on them (among which Riad). The GUI presents the semantics of the underlying sharing rule, that is a typical SWYSWYK reflexive rule represented as logic-based predicates on Cozy metadata. The GUI allows the attendee to identify that Riad has been granted access to certain pictures of the circle, confirmed when connecting to Riad’s personal cloud. The attendee can also share the GPS and activity trails of the circle with Riad very easily.

Step 3 - Sharing administration: the access control console allows the attendee (playing Paul’s role) to visualize and control the net effect of the current access control policy (set of all existing sharing rules). All resulting permissions are shown as viewable ACLs, i.e., triples $\langle \text{subject}, \text{object}, \text{permission} \rangle$ where each *subject* and *object* are personal cloud documents which can in turn be visualized. The GUI brings to light a suspicious permission that the attendee is invited to remove (or confirm according to her will).

Step 4 - Dynamicity: the demo operator finally selects from the Cozy the set of pictures taken during the conference, showing groups of people. Then, he takes a selfie with a demo attendee and creates her contact based on the photo, triggering her registration as a subject. This automatically grants her a read access on all the

²<https://cozy.github.io/cozy-stack/sharing.html>

conference pictures on which she appears, including the selfie and forthcoming ones, thanks to a pre-trained face recognition model.

3.3 Demonstration results

The demonstration shows an implementation of the SWYSWYK model in the Cozy platform. The semantics of the model and its administration principles, based on the combination of *the documents are rules and subjects and objects are documents* motto, opens to a set of benefits:

Ease-of-use. The content of a personal cloud, which describes Alice’s acquaintances, and conversely, acquaintances which are associated with pieces of information of Alice, are used to express sharing rules. Most interesting rules could also be easily shared via the Cozy marketplace, and reused among interested users.

Self-administration. The sharing rules are self-administrated while the personal cloud content evolves. Typically, new subjects (attendees) are automatically created while inserting new contact files or address book entries and a search of correspondences with potential content to share with them is automatically triggered.

Visualization. Subjects and objects are all viewable documents of the personal cloud. Hence the net effect of any sharing policy can be visualized and precisely apprehended by Alice (e.g., the GPS tracks pictured in a map that she is ready to share with a subject represented by her identity picture).

Control. Administration tools are provided to ease the detection of suspicious permissions and sanitize the access control policy. Pursuing this objective, watchdog triggers highlight newly generated ACLs involving sensitive subjects, documents and the associations of presumed incompatible subject/object pairs.

4 CONCLUSION

Finding new ways for the individual to intuitively share personal data and apprehend the real effects of their sharing policies is paramount. This is particularly true in a ubiquitous context where highly sensitive personal data (e.g., well-being data, daily activity logs) are produced at an increasing rate by smart appliances. Gathering these data in a personal cloud allows the definition

of new transversal services of great value for the individual and holds the promise of a better privacy than storing them in a central cloud. However, appropriate sharing tools are needed to regulate data sharing and prevent individuals from exposing their digital life because of too permissive sharing policies. This demonstration shows how the SWYSWYK model tackles this challenge. We hope that this work contributes to a new step in the privacy preservation of personal data.

REFERENCES

- [1] Serge Abiteboul, Benjamin André, and Daniel Kaplan. 2015. Managing your digital life. *Commun. ACM* 58, 5 (2015), 32–35.
- [2] Elisa Bertino, Gabriel Ghinita, Ashish Kamra, and others. 2011. Access control for databases: concepts and systems. *Foundations and Trends® in Databases* 3, 1–2 (2011), 1–148.
- [3] Barbara Carminati, Elena Ferrari, and Andrea Perego. 2006. Rule-based access control for social networks. In *OTM Confederated International Conferences "On the Move to Meaningful Internet Systems"*. Springer, 1734–1744.
- [4] Lujun Fang and Kristen LeFevre. 2010. Privacy wizards for social networking sites. In *Proceedings of the 19th international conference on World wide web*. ACM, 351–360.
- [5] Michelle L Mazurek, Yuan Liang, William Melicher, Manya Sleeper, Lujo Bauer, Gregory R Ganger, Nitin Gupta, and Michael K Reiter. 2014. Toward strong, usable access control for shared distributed data. In *Proceedings of the 12th USENIX conference on File and Storage Technologies*. USENIX Association, 89–103.
- [6] Jaehong Park and Ravi Sandhu. 2004. The UCON ABC usage control model. *ACM Transactions on Information and System Security (TISSEC)* 7, 1 (2004), 128–174.
- [7] Anna Cinzia Squicciarini, Smitha Sundareswaran, Dan Lin, and Josh Wede. 2011. A3p: adaptive policy prediction for shared images over popular content sharing sites. In *Proceedings of the 22nd ACM conference on Hypertext and hypermedia*. ACM, 261–270.
- [8] Amin Tootoonchian, Stefan Saroiu, Yashar Ganjali, and Alec Wolman. 2009. Lockr: better privacy for social networks. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies*. ACM, 169–180.
- [9] Paul Tran-Van, Nicolas Ancaix, and Philippe Pucheral. 2017. A new sharing paradigm for the personal cloud. In *International Conference on Trust and Privacy in Digital Business*. Springer, 180–196.
- [10] Paul Tran-Van, Nicolas Ancaix, and Philippe Pucheral. 2017. SWYSWYK: a new Sharing Paradigm for the Personal Cloud. In *International Conference on Advanced Data Mining and Applications*. Springer, 839–845.
- [11] Paul Tran-Van, Nicolas Ancaix, and Philippe Pucheral. 2017. SWYSWYK: a privacy-by-design paradigm for personal information management systems. In *International Conference on Information Systems Development (ISD)*.
- [12] Lingyu Wang, Duminda Wijesekera, and Sushil Jajodia. 2004. A logic-based framework for attribute based access control. In *Proceedings of the 2004 ACM workshop on Formal methods in security engineering*. ACM, 45–55.

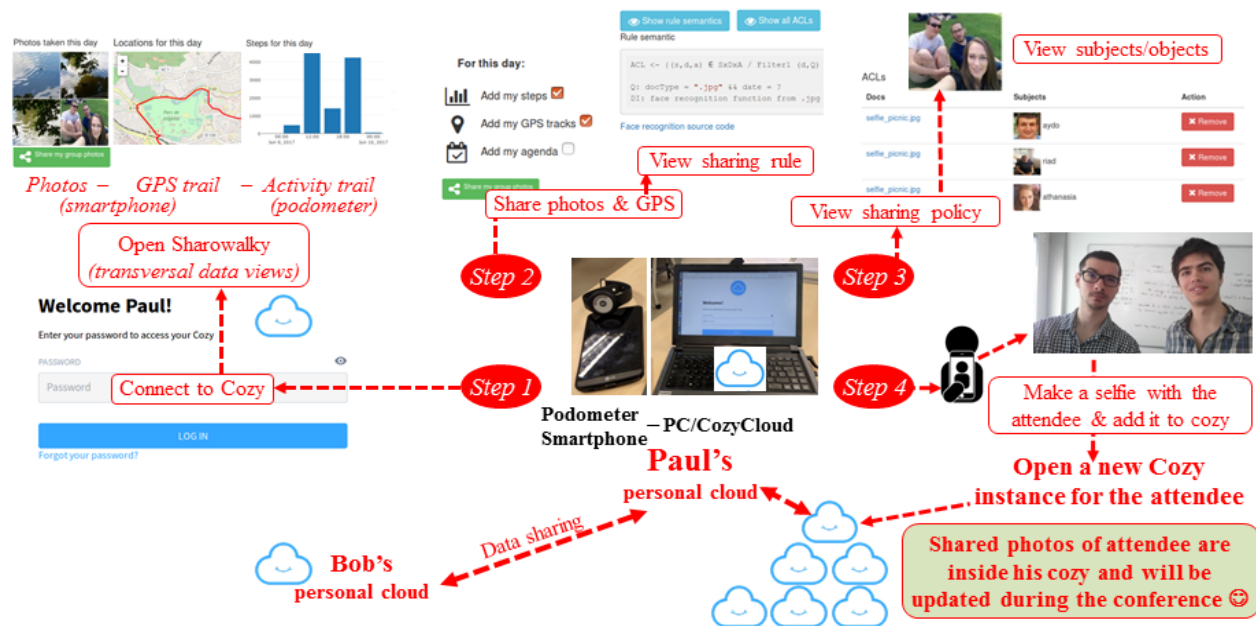


Figure 3: Demonstration scenario.

Spatio-Temporal-Keyword Pattern Queries over Semantic Trajectories with Hermes@Neo4j

Fragkiskos Gryllakis
 Dept. of Informatics
 University of Piraeus
 Piraeus, Greece
 fgryllakis@unipi.gr

Nikos Pelekis
 Dept. of Statistics and Ins. Science
 University of Piraeus
 Piraeus, Greece
 npelekis@unipi.gr

Christos Doulkeridis
 Dept. of Digital Systems
 University of Piraeus
 Piraeus, Greece
 cdoulk@unipi.gr

Stylianos Sideridis
 Dept. of Informatics
 University of Piraeus
 Piraeus, Greece
 ssider@unipi.gr

Yannis Theodoridis
 Dept. of Informatics
 University of Piraeus
 Piraeus, Greece
 ytheod@unipi.gr

ABSTRACT

In this paper, we demonstrate Hermes@Neo4j¹, an extension of Neo4j graph DBMS for semantic trajectories of moving objects, on the so-called Spatio-Temporal-Keyword Pattern queries. For this purpose, our engine exploits on hybrid Spatio-Temporal-Keyword (STK) index structures, also boosted by an appropriate selectivity estimation model. Hermes@Neo4j functionality is demonstrated over synthetic and real semantic trajectory datasets.

1 INTRODUCTION

The efficient management and analysis of the spatio-temporal evolution of a moving object (the so-called object's trajectory) has led to the development of plenty of appropriate index structures and algorithms, and even extensions over DBMS during the last two decades [1-5]. Recently, the research community has turned its interest to semantic trajectories [6], where spatio-temporal information is enriched with related annotations about the what, how, and why of movement [6-8].

The paradigm of Location-based Social Networking (LBSN) services, such as Twitter, Instagram and Foursquare, is indicative of this shift: the management and analytics over large amounts of spatio-temporal-textual data may result in useful conclusions about the users' behaviour.

Our motivation in this work is to demonstrate how a

Semantic Trajectory Database (STD), built on top of an extensible DBMS, can efficiently support queries where constraints are set over the triple (spatial, temporal, textual) nature of semantic trajectories. In particular, we aim to demonstrate the functionality of our Hermes@Neo4j STD engine over Spatio-Temporal-Keyword Pattern (STKP) queries [9].

According to [9], an STKP query is defined as follows: Let E_i be a semantic trajectory episode abstraction, which is defined as a partially or completely defined episode. An episode abstraction therefore is an episode where some of its properties – spatial, temporal, textual information – may be missing. An STKP query over a STD takes as input a sequence Q of episode abstractions or the * wildcard (more formally, $Q := \langle p^* \mid p \text{ is either an episode abstraction } E_i \text{ or the } * \text{ wildcard} \rangle$) and gives as output the semantic trajectories in STD that are compatible with Q .

An example of STKP query follows in Fig. 1 [9].

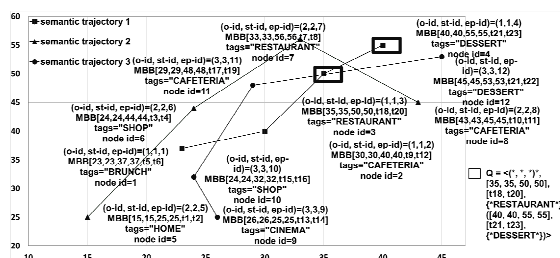


Figure 1: Graph representation of an STD consisting of 3 trajectories along with a STKP query.

In Fig. 1, we depict a STD consisting of 3 semantic trajectories; each trajectory consists of four episodes. An example STKP query Q is also illustrated at the bottom right corner. In particular, Q consists of a number of episode abstractions; with notation E_i^* corresponding to a number of

¹More information regarding Hermes@Oracle and Hermes@Postgres are available at www.datastories.org/hermes.

zero or more episode abstractions of the form E_i . For clarity of presentation the episode abstractions in Q distinguish the temporal from the spatial information, which is not the case in reality where both are organized together in a Minimum Bounding Box (MBB). Actually, Q searches for trajectories starting with zero or more episodes of any kind (see notation $(*, *, *)^*$ in Q), followed by an episode in a spatial [35, 35, 50, 50] and temporal $[t_{18}, t_{20}]$ range with keyword 'RESTAURANT' and ending with an episode in a spatial [40, 40, 55, 55] and temporal $[t_{21}, t_{23}]$ range with keyword 'DESSERT'. The output set includes semantic trajectory 1, which fulfills the above constraints.

The practical contribution of this work is that we present a framework that utilizes recently introduced (a) state-of-the-art hybrid indexes and (b) query processing algorithms on (c) a new STD engine, coined Hermes@Neo4j STD engine.

Hermes@Neo4j STD engine provides efficient and effective storage, indexing mechanisms and a library of utilities that facilitate spatio-temporal and textual operations on data, able to support STDs. More specifically, the merits and contributions of Hermes@Neo4j STD engine are summarized below:

1. Following the successful MOD engine paradigm of Hermes@Oracle [7,8], we designed a new datatype system for the representation and management of Semantic Trajectories into the extensible DBMS architecture of Neo4j [10], an ACID-compliant transactional NoSQL DBMS with native graph storage and processing, implemented in Java. The datatype system is formulated in the context of the graph database, that provides an intuitive model in our case.
2. Neo4j Spatial library [11], which is a library of utilities for Neo4j that facilitates spatial operations on data, is extended for processing the spatio-temporal and textual information of semantic trajectories.
3. We designed efficient access methods for semantic trajectories, called TSR-tree and TSR-tree indexes, for the hybrid indexing of both the spatio-temporal and the textual component. The hybrid indexes combine a spatiotemporal and text index tightly, such that both types of information can be used to prune the search space simultaneously during the spatiotemporal keyword query algorithm processing in STDs.
4. We developed efficient query processing algorithms upon the proposed indices in order to support a useful query type at the STD level, called STKP, as well as algorithms for efficiently importing semantic trajectories into STDs.

Employing separate indexes is weaker in comparison with the tightly integrated proposed state-of-the-art approach, since an efficient resolution of the STKP requires repetitive invocation of spatio-temporal-keyword matching queries.

The paper is organized as follows: in Section 2 we provide a brief presentation of the system architecture, the underlying indexes, etc. (the interested reader is referred to [9] for more details). Section 3 provides more information about system implementation details. Finally, Section 4 outlines the flow of the demonstration.

2 SYSTEM ARCHITECTURE

In this section, we present the core information about the architecture of our framework, illustrated in Fig. 2.

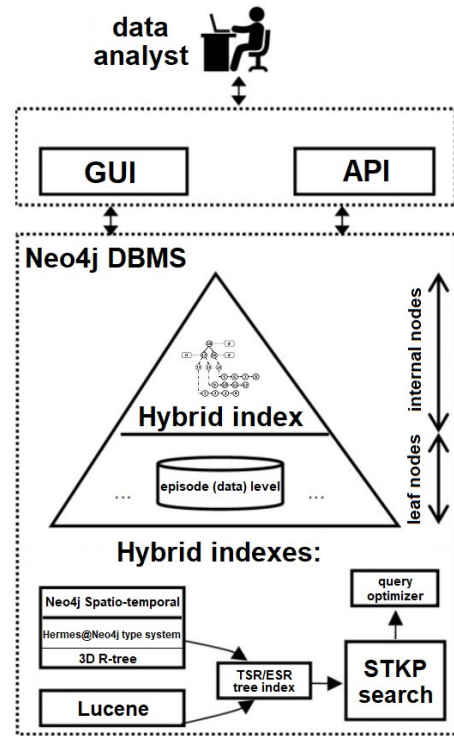


Figure 2: Hermes@Neo4j STD engine architecture.

2.1 Hybrid indexes

Neo4j Spatial R-tree index is extended to support spatio-temporal and classical trajectory-based queries (3D R-Tree) and is also used in order to enable effective spatiotemporal-keyword operations on semantic trajectories. There are two alternative indexing structures to support efficient STKP query processing. The proposed hybrid indices combine tightly a spatial and a text index (i.e. a 3D-Rtree and inverted file, respectively), so that both types of information can be used simultaneously for pruning the search space.

TSR-tree index. In this index, a semantic trajectory is considered as an individual unit for the tree construction. More specifically, for each semantic trajectory we compute its MBB and a list of tags related to the semantics of the episodes, sorted by time. MBB is the minimum bounding box that encloses a specific sub-trajectory of a moving object, along with the start and end times of the movement. At the end, the tags in the list are concatenated to a single string. Specifically, a pseudo-word for each semantic trajectory is created with all the concatenated tags of each trajectory's episode to a single string in order to use it for the keyword query search criteria. The leaves of the TSR-tree index are the above-described approximation of the whole semantic trajectories. For the exploitation of the graph database where our index resides, these leaf nodes are the starting nodes

of the sequence of the episodes of the approximated semantic trajectory. Moreover, inverted files (IFs) are created for all the internal nodes of the tree.

ESR-tree index. As an alternative, we build a tree using as its structural unit the episodes of the semantic trajectories rather than the semantic trajectories themselves. In other words, ESR-tree index takes into account as structural unit for the creation of the 3D-Rtree the episodes of the semantic trajectories. The leaves of the ESR-tree index are the base for the creation of the episode MBBs. Accordingly, the MBBs of the entries of the internal nodes represent the spatio-temporal union of the MBBs of the children nodes. Additionally, for each internal node there exists a pointer to an IF that organizes all the tags of its children nodes. The IFs for each internal node of the tree contain the keywords of the episodes of its child nodes.

2.2 STKP query processing

STKP queries can be processed in Hermes@Neo4j using either TSR-tree or ESR-tree index. In the former (latter) approach, STKP search algorithm takes into account that TSR-tree (ESR-tree, respectively) is built using entire semantic trajectories (episodes of semantic trajectories, respectively) as building blocks [9]. It is noteworthy that STKP search is boosted by an optimization method.

STKP query optimizer. Given a STKP query $Q := \langle E_1, \dots, E_k \rangle$, where E_1, \dots, E_k is a sequence of spatio-temporal-textual constraints over episode abstractions, the STKP query optimizer identifies the most selective episode abstraction E^* in Q , in order to start the execution of the ESR-tree search algorithm from there, thereby pruning candidate results the earliest possible. The cost model that the query optimizer implements decomposes the computation of selectivity of an episode abstraction in two parts, one for the spatio-temporal and another for the textual component of the episode abstraction [9].

Regardless of the query length, it turns out that the search based on the ESR-tree and boosted by the query optimizer, is considerably faster, with the penalty of the higher index creation time and size compared with the TSR-tree approach.

3 SYSTEM IMPLEMENTATION

Our framework provides a robust API with the necessary tools for STD creation, querying, etc. Toward the realization of the concepts of semantic trajectories and STDs presented in the previous section, we followed the object relational (OR) approach for the datatype system of Hermes@Neo4j STD engine. In detail, we follow the abstract datatype (ADT) paradigm and define the episodes and semantic trajectory datatypes that support the definitions in [9]. Upon these datatypes, we register a rich palette of object methods; some indicative examples appear in Table 1. More details are available at Hermes@Neo4j web page².

Table 1: Methods over episodes and semantic trajectories

² <http://infolab.cs.unipi.gr/HermesNeo4j/>

Object	Method	Definition
Episode	duration()	Returns the episode duration.
Semantic Trajectory	num_of_episodes (String tag, String distinct), where “tag” is a set of substrings and a boolean string.	Returns the number of episodes (distinct or not, depending on distinct string) that includes tags LIKE the given ones (pattern-matching per input tag).
LayerST (object with the episodes and semantic trajectories of an STD)	confined_in (LayerST layer, MBB envelope, String tag), where tag is a concatenated set of substrings.	Returns semantic trajectories, whose episodes are overlapping spatiotemporally with the MBB and textually with the “tag” parameter.

STD creation. STD construction includes two phases (3D R-tree and IFs), along with segmental options for creating a graph database in steps (separate 3D R-tree and IFs creation routines) in case of size and memory concerns.

Semantic Trajectory Synthesizer. In case of trajectory datasets lacking textual annotations, our synthesizer is able to augment raw trajectories with textual annotations using a customized text generator that chooses terms from a lexicon of V keywords. The number of keywords for each episode follows a Zipfian distribution, in order to simulate the skewness present in real-life textual datasets.

STD search. Several functions are available for a wide range of queries like intersection, overlapping or union, with the emphasis, of course, in STKP queries (details for the appropriate search methods appear in Table 2).

Table 2: STKP query methods

Method	Parameters	Index
SpatialTemporalKeywordTrajectoryQuery	LayerST, list of MBBs,	TSR-tree index
SpatialTemporalKeywordTrajectoryEpisodesQuery	list of String tags	ESR-tree index

Hermes@Neo4j STD engine utilizes Apache Lucene [12] indexes that use inverted indexes for search and retrieval from text collections. For the implementation of interactive visualizations of the semantic trajectories over a 3D model of the globe and different types of 2D maps, the NASA WorldWind API [13] is utilized. The library has been extended in order to display the spatio-temporal and textual information of a semantic trajectory. Visual representation of search results is performed through different 3D / 2D map services, such as Open Street Map, Bing, MS Virtual Earth, NASA Blue Marble and i-cubed Landsat (Fig. 3).

The interface has the required parameters for spatio-temporal

and textual constraints that are used as query arguments. The interface also includes necessary options for importing a dataset to a new STD. Apart from setting spatio-temporal and textual constraints, in order to perform a STKP query over the selected STD, the user decides the index and search algorithm of his/her choice. The semantic trajectories that are the results of the STKP query are displayed through a proper animation zoom at the selected map service and reference system by displaying the geographical area that covers the specific semantic trajectories. Correspondingly, information about the search results and the number of trajectories that meet the search criteria are displayed in a relevant result box.

4 ABOUT THE DEMONSTRATION

Throughout the demonstration, Hermes@Neo4j users will be able to test the system by using the real “Foursquare New York” dataset [14] and the synthetic “Hermes Attica” dataset³ generated by the Hermoupolis generator [15]. The real dataset includes long-term (about 10 months - from Apr.12, 2012 to Feb.16, 2013) check-in data (227,428 check-ins) in New York City collected from Foursquare social network and the synthetic dataset consists of a total of 1,450,738 records that represent semantic trajectories.



Figure 3: Interactive visual exploration of a semantic trajectory that is the result of an STKP query through a 2D map representation.

The demonstration captures the following phases: (i) preparatory phase, where users have the opportunity to comprehend the internals of our framework, and (ii) our Hermes@Neo4j engine in action, where users experience various scenarios of STKP search. In particular:

Preparatory phase (background knowledge). During this phase, we show off the different datatypes and operands that can be utilized in the Hermes@Neo4j engine. In addition, we demonstrate how the user can use our framework to run legacy operands, and even more interestingly, focus on the two STKP query indexing and searching approaches.

Hermes@Neo4j engine in action. Having gained the necessary background knowledge, the user experiences a scenario of STKP search and index creation, based on the TSR-tree search algorithm. For instance, Fig. 3 present an example of

an STKP query result, which is a semantic trajectory from the “Hermes Attica” dataset. The user can display the results with a selected 3D/2D map representation and reference system of his/her choice. In addition, the user can interactively switch on and off the visibility of the results. In turn, we present a scenario of STKP search and index creation, based on the ESR-tree search algorithm. The goal of this scenario is to effectively demonstrate that the STKP search based on the ESR-tree, is more efficient in comparison with the STKP search based on the TSR-tree index, with the penalty of the higher index creation time and size compared with the TSR-index.

For a deeper comprehension of the demonstrated functionality, a related video is available at Hermes@Neo4j web page⁴.

ACKNOWLEDGEMENTS

This work has been partly supported by the University of Piraeus Research Center. This project has received funding from the European Union’s Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie agreement N. 777695.

REFERENCES

- [1] Nikos Pelekis, Elias Frentzos, Nikos Giatrakos, and Yannis Theodoridis. 2008. HERMES: aggregative LBS via a trajectory DB engine. In Proceedings of SIGMOD. <https://doi.org/10.1145/1376616.1376748>
- [2] Ralf H. Güting, Thomas Behr, and Christian Düntgen. 2010. SECONDO: a platform for moving objects database research and for publishing and integrating research implementation. IEEE Data Engineering Bulletin, 33(2):56-63.
- [3] Cédric du Mouza and Philippe Rigaux. 2005. Mobility patterns. GeoInformatica, 9 (4): 297-319. <https://doi.org/10.1007/s10707-005-4574-9>
- [4] Marcos R. Vieira, Petko Bakalov, and Vassilis J. Tsotras. 2010. Querying trajectories using flexible patterns In: Proc. of EDBT. <https://doi.org/10.1145/1739041.1739091>
- [5] Ralf H. Güting, Fabio Valdés, and Maria L. Damiani. 2015. Symbolic Trajectories. ACM Transactions on Spatial Algorithms and Systems, 1(2). <https://doi.org/10.1145/2786756>
- [6] Christine Parent, Stefano Spaccapietra, Chiara Renso, Gennady Andrienko, Natalia Andrienko, Vania Bogorny, Maria L. Damiani, Aris Gkoulalas-Divanis, Jose Macedo, Nikos Pelekis, Yannis Theodoridis, and Zhixian Yan. 2013. Semantic Trajectories Modeling and Analysis. ACM Computing Surveys, 45(4), article 42. <https://doi.org/10.1145/2501654.2501656>
- [7] Nikos Pelekis, Stylianos Sideridis, and Yannis Theodoridis. 2015. Hermes^{sem}: a Semantic-aware Framework for the Management and Analysis of our LifeSteps. In: Proc. of DSAA. <https://doi.org/10.1109/DSAA.2015.7344849>
- [8] Stylianos Sideridis, Nikos Pelekis, and Yannis Theodoridis. 2016. On querying and mining semantic-aware mobility timelines. International Journal of Data Science and Analytics, 2(1-2), 29-44 (2016). <https://doi.org/10.1007/s41060-016-0030-1>
- [9] Fragkiskos Gryllakis, Nikos Pelekis, Christos Doukeridis, Stylianos Sideridis, and Yannis Theodoridis. 2017. Searching for Spatio-Temporal-Keyword Patterns in Semantic Trajectories. In: Proc. of IDA. https://doi.org/10.1007/978-3-319-68765-0_10
- [10] Neo4j graph database, <https://neo4j.com/>.
- [11] Neo4j Spatial library, <http://neo4j-contrib.github.io/spatial/>.
- [12] Apache Lucene, <https://lucene.apache.org/>.
- [13] NASA WorldWind, <https://worldwind.arc.nasa.gov/>.
- [14] Dingqi Yang, Daqing Zhang, Vincent W. Zheng, and Zhiyong Yu. 2015. Modeling User Activity Preference by Leveraging User Spatial Temporal Characteristics in LBSNs. TSMC, 45(1). <https://doi.org/10.1109/TSMC.2014.2327053>
- [15] Nikos Pelekis, Stylianos Sideridis, Panagiotis Tampakis, and Yannis Theodoridis. Simulating our LifeSteps by Example (2016) ACM Trans. Spatial Algorithms and Systems, 2(3), article 11. <https://doi.org/10.1145/2937753>

³ <http://chorochronos.datastories.org/?q=content/hermes-attica>

⁴ <http://infolab.cs.unipi.gr/HermesNeo4j/visual.htm>

MDM: Governing Evolution in Big Data Ecosystems

<p>Sergi Nadal Universitat Politècnica de Catalunya Barcelona, Spain snadal@essi.upc.edu</p>	<p>Alberto Abelló Universitat Politècnica de Catalunya Barcelona, Spain aabello@essi.upc.edu</p>	<p>Oscar Romero Universitat Politècnica de Catalunya Barcelona, Spain oromero@essi.upc.edu</p>
---	---	---

<p>Stijn Vansummeren Université Libre de Bruxelles Brussels, Belgium svsummer@ulb.ac.be</p>	<p>Panos Vassiliadis University of Ioannina Ioannina, Greece pvassil@cs.uoi.gr</p>
--	---

ABSTRACT

On-demand integration of multiple data sources is a critical requirement in many Big Data settings. This has been coined as the data variety challenge, which refers to the complexity of dealing with an heterogeneous set of data sources to enable their integrated analysis. In Big Data settings, data sources are commonly represented by external REST APIs, which provide data in their original format and continuously apply changes in their structure (i.e., schema). Thus, data analysts face the challenge to integrate such multiple sources, and then continuously adapt their analytical processes to changes in the schema. To address this challenges, in this paper, we present the Metadata Management System, shortly MDM, a tool that supports data stewards and analysts to manage the integration and analysis of multiple heterogeneous sources under schema evolution. MDM adopts a vocabulary-based integration-oriented ontology to conceptualize the domain of interest and relies on *local-as-view* mappings to link it with the sources. MDM provides user-friendly mechanisms to manage the ontology and mappings. Finally, a query rewriting algorithm ensures that queries posed to the ontology are correctly resolved to the sources in the presence of multiple schema versions, a transparent process to data analysts. On-site, we will showcase using real-world examples how MDM facilitates the management of multiple evolving data sources and enables its integrated analysis.

1 INTRODUCTION

In recent years, a vast number of organizations have adopted data-driven approaches that align their business strategy with advanced data analysis. Such organizations leverage Big Data architectures that support the definition of complex data pipelines in order to process heterogeneous data, from multiple sources, in their original format. External data (i.e., neither generated nor under control of the organization) are commonly ingested from third party data providers (e.g., social networks) via REST APIs with a fixed schema. This requires data analysts to tailor their processes to the imposed schema for each source. A second challenge that data analysts face is the adaptation of such processes upon schema changes (i.e., a release of a new version of the API), a cumbersome task that needs to be manually dealt with. For instance, in the last year Facebook’s Graph API¹ released four major versions

affecting more than twenty endpoints each, many of them breaking changes. The maintenance of such data analysis processes is critical in scenarios integrating tenths of sources and exploiting them in hundreds of analytical processes, thus its automation is badly needed.

The definition of an integrated view over an heterogeneous set of sources is a challenging task that Semantic Web technologies are well-suited for to overcome the *data variety* challenge [3]. Given the simplicity and flexibility of ontologies, they constitute an ideal tool to define a unified interface (i.e., global vocabulary or schema) for such heterogeneous environments. This family of systems, that perform data integration using ontologies, propose to define a global conceptual schema (i.e., by means of an ontology) over the sources (i.e., by means of mappings) in order to rewrite ontology-mediated queries (OMQs) to the sources. The state of the art approaches for such integration-oriented ontologies are based on generic reasoning algorithms, that rely on certain families of description logics (DLs). Such approaches rewrite an OMQ, first to an expression in first-order logic and then to SQL. This approach, commonly referred as *ontology-based data access* (OBDA) [8], does not consider the management of changes in the sources, and thus such variability in their schema would cause OMQs either crash or return partial results. This issue, which is magnified in Big Data settings, is caused because OBDA approaches represent schema mappings following the *global-as-view* (GAV) approach, where elements of the ontology are characterized in terms of a query over the source schemata. GAV ensures that the process of query rewriting is tractable and yields a first-order logic expression, by just unfolding the queries to the sources, but faulty upon source schema changes [2]. To overcome this issues a desiderata is to adopt the *local-as-view* (LAV) approach. Oppositely to GAV, LAV characterizes elements of the source schemata in terms of a query over the ontology, making it inherently more suitable for dynamic environments [4]. LAV flexibility, however, comes at the expense of computational complexity in the query answering process.

To address these challenges, we adopt a vocabulary-based approach for data integration. These approaches are not necessarily restricted to the expressiveness of a DL and its generic reasoning algorithms. Such settings rely on rich metamodels for specific integration tasks, here focused on schema evolution. Under certain constraints when instantiating the metamodel, it is possible to define specific efficient algorithms that resolve LAV mappings without ambiguity. To this end, we created the Metadata Management System, or shortly MDM², an end-to-end solution to assist data stewards and data analysts during the Big Data integration lifecycle. Data stewards are provided with mechanisms to

¹<https://developers.facebook.com/docs/graph-api/changelog>

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

²<http://www.essi.upc.edu/~snadal/mdm.html>

semi-automatically integrate new sources and accommodate schema evolution into a global schema. In turn, data analysts have means to pose OMQs to such global schema by making transparent the underlying mechanisms to query the sources with LAV mappings.

MDM implements a vocabulary-based integration-oriented ontology, represented by means of two RDF graphs, specifically the global graph and the source graph [5]. The former representing the domain of interest (also known as domain ontology) and the latter the schema of the sources. The key concepts are *releases*, which represent a new source or changes in existing sources. A relevant element of releases are *wrappers* (from the well-known mediator/wrapper architecture in data integration), the mechanism enabling access to the sources (e.g., an API request or a database query). Upon new releases the schemata of wrappers are extracted and their RDF-based representation stored in the source graph. Afterwards, the data steward is aided on the process of linking such new schemata to the global graph (i.e., define the LAV mapping). Orthogonally, data analysts pose OMQs to the global graph. The current de-facto standard to query ontologies is the SPARQL query language, however to enable non-expert analysts to query the sources MDM offers an interface where OMQs are graphically posed as subgraph patterns of the global graph, which are automatically translated to SPARQL. A specific query rewriting algorithm takes care of how to properly resolve LAV mappings, a process that consists on the discovery of joins amongst wrappers and their attributes, regardless of the number of wrappers per source.

Motivational use case. As motivational use case, and for the sake of understandability, we will analyse information related to European football teams. This represents the simple use case that will be demoed on-site amongst others with higher complexity (i.e., the SUPERSEDE project). Precisely, we aim to ingest data from four data sources, in the form of REST APIs, respectively providing information about players, teams, leagues and countries. The integrated schema of this scenario is conceptualized in the UML depicted in Figure 1, which we use as a starting point to provide a high-level representation of the domain of interest, used to generate the ontological knowledge captured in the global graph.

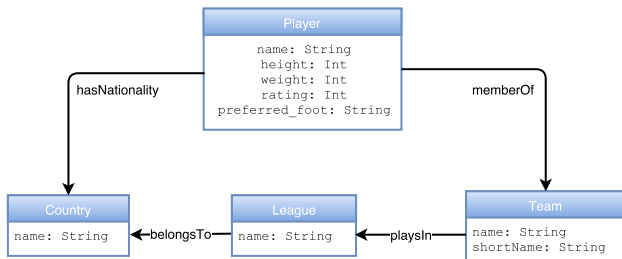


Figure 1: UML of the motivational use case

Each of the APIs is independent from each other, and thus they differ in terms of schema and format. Thus, for instance, the *Players API* provides data in JSON format while the *Teams API* in XML. An excerpt of the content provided by such two APIs is depicted in Figure 2. Next, the goal is to enable data analysts to pose OMQ to the ontology-based representation of the UML diagram (i.e., global graph) by navigating over the classes. Specifically, we aim the sources to be automatically accessed under multiple schema versions. An exemplary query would be, “*who are the players that play in a league of their nationality?*”.

Outline. In the rest of the paper, we will introduce the demonstrable features to resolve the motivational and other exploratory

```

{
  "id": 6176,
  "name": "Lionel Messi",
  "height": 170.18,
  "weight": 159,
  "rating": 94,
  "preferred_foot": "left",
  "team_id": 25
}

```

```

<team>
  <id>25</id>
  <name>FC Barcelona</name>
  <shortName>FCB</shortName>
</team>

```

Figure 2: Sample data for *Players API* and *Teams API*

queries. We first provide an overview of MDM and then, we present its core features to be demonstrated. Lastly, we outline our on-site presentation, involving the motivational use case and a complex real-world use case.

2 DEMONSTRABLE FEATURES

MDM presents an end-to-end solution to integrate and query a set of continuously evolving data sources. Figure 4 depicts a high-level overview of the approach. Its pillar is the Big Data integration (BDI) ontology [7], the metadata model (i.e., set of design guidelines) that allow data stewards to semantically annotate the integration constructs that enable automating the evolution process and unambiguously resolve query answering.

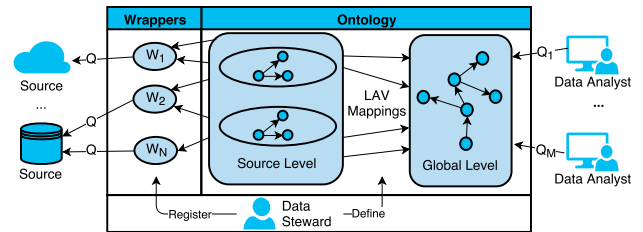


Figure 4: High-level overview of our approach

We devise four kinds of interaction with the system, which are in turn the offered functionalities: (a) *definition of the global graph*, where data stewards define the domain of interest for analysts to query; (b) *registration of wrappers*, either in the presence of a new source or the evolution of an existing one; (c) *definition of LAV mappings*, where LAV mappings between the source and the global graphs are defined; and (d) *querying the global graph*, where data analysts pose OMQs to the global graph which are automatically rewritten over the wrappers. In the following subsections, we describe how MDM assists on each of them.

2.1 Definition of the global graph

The global graph, whose elements are prefixed with G, reflects the main domain concepts, relationships among them and features of analysis. To this end, we distinguish between two main constructs *concepts* and *features*. Concepts (i.e., instances of $G:Concept$) are elements that group features (i.e., $G:Feature$) and do not take concrete values from the sources. Only concepts can be related to each other using any user-defined property, we also allow to define taxonomies for them (i.e., $rdfs:subClassOf$). It is possible to reuse existing vocabularies to semantically annotate the data at the global graph, and thus follow the principles of Linked Data. This, enables data to be self-descriptive as well as it opens the door to publish it on the Web [1]. Furthermore, we restrict features to belong to only one concept.

MDM supports the definition of the global graph avoiding the need to use external ontology modeling tools (e.g., *Protégé*). Figure 5 depicts an excerpt of the global graph for the demo use case, focusing on the concepts *Player* and *Team*. Like we said, we reuse vocabularies as much as possible, hence the concept *Team* is reused from <http://schema.org/SportsTeam>. When no reuse is possible, we define the example’s custom prefix *ex*. As data stewards interact with MDM to define the global graph, the corresponding RDF triples are being generated automatically.

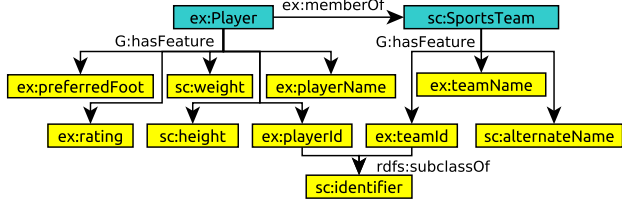


Figure 5: Global graph for the motivational use case. Blue and yellow nodes denote concepts and features

2.2 Registration of new data sources

New wrappers are introduced either because we want to consider data from a new data source, or because the schema of an existing source has evolved. Nevertheless, in both cases the procedure to incorporate them to the source level, whose elements are prefixed with *S*, is the same. To this end, we define the data source (i.e., *S:DataSource*) and wrapper (i.e., *S:Wrapper*) metaconcepts. Data stewards must provide the definition of the wrapper, as well as its signature. We work under the assumption that wrappers provide a flat structure in first normal form, thus the signature is an expression of the form $w(a_1, \dots, a_n)$ where w is the wrapper name and a_1, \dots, a_n the set of attributes. With such information, MDM extracts the RDF-based representation of the wrapper’s schema (i.e., creates elements of type *S:Attribute*) which are incorporated to the existing source level. In the case of a wrapper for an existing data source, MDM will try to reuse as many attributes as possible from the previous wrappers for that data source. However, this is not possible among different data sources as the semantics of attributes might differ. In the case of attributes in the source graph, as they are not meant to be shared, oppositely to features in the global graph, there is no need to reuse external vocabularies.

Figure 6 depicts an excerpt of the source graph for the sources related to players and teams, the former with a wrapper’s signature $w_1(id, pName, height, weight, score, foot, teamId)$ and the latter $w_2(id, name, , shortName)$. Note that, for w_1 , some attribute names differ from the data stored in the source (see Figure 2), this is due to the fact that the query contained in the wrapper might rename (e.g., *foot*) or add new attributes (e.g., *teamId*). The definition of a wrapper (e.g., a MongoDB query, a Spark job, etc.) is out of the scope of MDM and should be carried out by the data steward.

2.3 Definition of LAV mappings

LAV mappings are encoded as part of the ontology. We represent them as two components, (a) a subgraph of the global graph, one per wrapper, and (b) a function linking attributes from the source graph to features in the global. The former are achieved thanks to RDF named graphs, which allow to identify subsets of other RDF

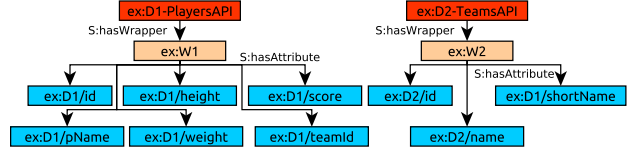


Figure 6: Source graph for the motivational use case. Red, orange and blue denote data sources, wrappers and attributes

graphs identified by an URI. In this case, the URI will be the one for the wrapper. The latter are achieved via the `owl:sameAs` property. Note that, traditionally, the definition of LAV mappings was a difficult task even for IT people. However, in MDM LAV mappings can be easily asserted through the interface: each wrapper must map to a named graph (i.e., a subset of the global graph), and a set of `owl:sameAs` from attributes to features. The task consists on first selecting a wrapper, and then, with the mouse, drawing a contour around the set of elements in the global graph that this wrapper is populating (including concept relations).

Figure 7 depicts the LAV mappings for wrappers w_1 and w_2 , respectively in red and green. Note the intersection in the concept `sc:SportsTeam` and its identifier, this will be later used when querying in order to enable joining such concepts. However, this joins are only restricted to elements that inherit from `sc:identifier`.

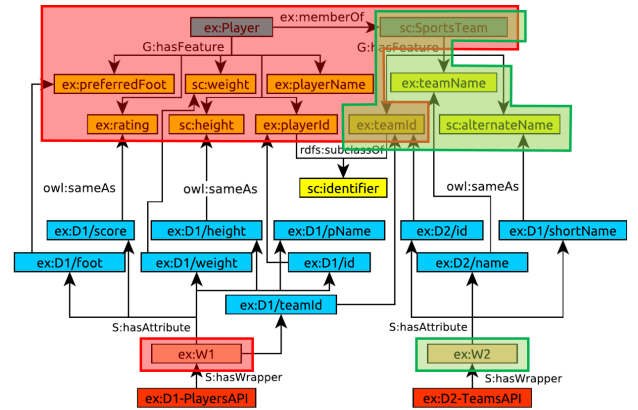


Figure 7: LAV mappings for the motivational use case

2.4 Querying the global graph

To overcome the complexity of writing SPARQL queries over the global graph, MDM adopts a graph pattern matching approach to enable non-technical data analysts perform their OMQs. Recall that the *WHERE* clause of a SPARQL query consists of a graph pattern. To this end, the analyst can graphically select a set of nodes of the global graph representing such pattern, we refer to it as a *walk*. Then, a specific query rewriting algorithm takes as input a walk and generates as a result an equivalent union of conjunctive queries over the wrappers resolving the LAV mappings [7]. Such process consists of three phases: (a) *query expansion*, where the walk is automatically expanded to include concept identifiers that have not been explicitly stated; (b) *intra-concept generation*, that generates partial walks per concept indicating how to query the wrappers in order to obtain the requested features for the concept at hand; and (c) *inter-concept generation*, where all partial walks are joined to obtain a union of conjunctive queries.

Using the excerpt of the ontology depicted in Figure 7, we could graphically pose an OMQ fetching the name of the players and their teams. Figure 8 shows how such query can be defined in MDM by drawing a contour (in red) around the concepts and features of interest in the global graph. On the right hand side, it is depicted the equivalent SPARQL query, as well as the generated relational algebra expression over the wrappers. Table 1 depicts a sample of the output resulting of the execution of the query.

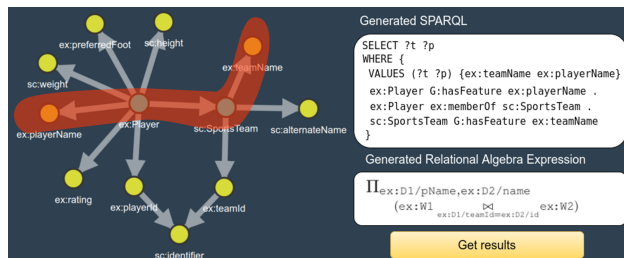


Figure 8: Posing an OMQ in MDM

ex:teamName	ex:playerName
FC Barcelona	Lionel Messi
Bayern Munich	Robert Lewandowski
Manchester United	Zlatan Ibrahimovic

Table 1: Sample output for the exemplary query.

2.5 Implementation details

MDM has been developed at UPC BarcelonaTech in the context of the SUPERSEDE³ project using a service-oriented architecture. It is the cornerstone of the Big Data architecture supporting the project, and a central component of its Semantic Layer [6]. On the frontend, MDM provides the web-based component to assist the management of the Big Data evolution lifecycle. This component is implemented in *JavaScript* and resides in a *Node.JS* web server. The interface makes heavy use of the *D3.js* library to render graphs and enables the user to interact with them. Web interfaces are defined using the *Pug* template engine, and a number of external libraries are additionally used. The backend is implemented as a set of REST APIs defined with *Jersey* for *Java*, thus the frontend interacts with the backend by means of HTTP REST calls. This enables extensibility of the system and a separation of concerns in such big system. The backend makes heavy use of *Jena* to deal with RDF graphs, as well as its persistence engine *Jena TDB*. Additionally, a *MongoDB* document store is responsible of storing the system’s metadata. Concerning the execution of queries, the fragment of data provided by wrappers is loaded into temporal SQLite tables in order to execute the federated query.

3 DEMONSTRATION OVERVIEW

In the on-site demonstration, we will present the functionality of MDM relying based on two use cases. First, we will focus on the paper’s motivational scenario, in order to comprehensively show the functionalities offered by MDM. Next we will focus on the SUPERSEDE use case, a real-world scenario of Big Data integration under schema evolution in order to show the full potential and benefits of MDM. We will cover the four possible kinds of interactions with MDM, taking the role of both data steward (definition of the global graph, registration of new wrappers, definition of LAV mappings) and data analyst (querying the global graph). We

³<https://www.supersede.eu>

will showcase how MDM aids on each of the processes, considering as well the input from participants. Precisely, the following scenarios will be covered:

System setup. In the first scenario we will take the role of a data steward that has been given a UML diagram (likewise Figure 1), and assigned the task of setting up a global schema to enable integrated querying of a disparate set of sources. Thus, we will show how MDM supports the definition of its equivalent global graph (likewise Figure 5) within the interface. Once finished, we will introduce the four sources (i.e., the players API, teams API, etc.) and a wrapper for each. We will show how MDM automatically extracts the schemata of wrappers to automatically generate the source graph (likewise Figure 6). Finally, we will show how MDM supports the graphical definition of named graphs, which are the basis for LAV mappings, and thus properly maps the source and global graphs (likewise Figure 7).

Ontology-mediated queries. With the global graph set up and a set of data sources and wrappers in place, now we can act as data analysts in order to pose OMQs to the system. We will encourage participants to propose their queries of interest, this is possible because MDM presents the global graph and allows to graphically draw a walk around its nodes. This is later automatically translated to its SPARQL form (likewise Figure 8), and to a relational algebra expression derived from the query rewriting process. MDM presents the execution of the query in tabular form.

Governance of evolution. In Big Data ecosystems, changes in the structure of the data sources will frequently occur. In this scenario, we will release a new version of one of the APIs including breaking changes that would cause the previously defined queries to crash. First, we will showcase how MDM easily supports the inclusion of this new source into the existing global graph and the definition of its LAV mappings. Next, we will execute again the queries that were supposed to crash showing how MDM has adapted the generated relational algebra expressions, where the two schema versions are now fetched and yield correct results.

ACKNOWLEDGMENTS

This work was partly supported by the H2020 SUPERSEDE project, funded by the EU Information and Communication Technologies Programme under grant agreement no 644018, and the GENESIS project, funded by the Spanish Ministerio de Ciencia e Innovación under project TIN2016-79269-R.

REFERENCES

- [1] Christian Bizer, Tom Heath, and Tim Berners-Lee. 2009. Linked Data - The Story So Far. *Int. J. Semantic Web Inf. Syst.* 5, 3 (2009), 1–22.
- [2] Loredana Caruccio, Giuseppe Polese, and Genoveffa Tortora. 2016. Synchronization of Queries and Views Upon Schema Evolutions: A Survey. *ACM Trans. Database Syst.* 41, 2 (2016), 9:1–9:41.
- [3] Ian Horrocks, Martin Giese, Evgeny Kharlamov, and Arild Waaler. 2016. Using Semantic Technology to Tame the Data Variety Challenge. *IEEE Internet Computing* 20, 6 (2016), 62–66.
- [4] Petar Jovanovic, Oscar Romero, and Alberto Abelló. 2016. A Unified View of Data-Intensive Flows in Business Intelligence Systems: A Survey. *T. Large-Scale Data- and Knowledge-Centered Systems* 29 (2016), 66–107.
- [5] Maurizio Lenzerini. 2002. Data Integration: A Theoretical Perspective. In *PODS*. 233–246.
- [6] Sergi Nadal, Victor Herrero, Oscar Romero, Alberto Abelló, Xavier Franch, Stijn Vansummeren, and Danilo Valerio. 2017. A software reference architecture for semantic-aware Big Data systems. *Information & Software Technology* 90 (2017), 75–92.
- [7] Sergi Nadal, Oscar Romero, Alberto Abelló, Panos Vassiliadis, and Stijn Vansummeren. 2018. An Integration-Oriented Ontology to Govern Evolution in Big Data Ecosystems. *Submitted to Information Systems* (2018). Available at <https://arxiv.org/abs/1801.05161>.
- [8] Antonella Poggi, Domenico Lembo, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. 2008. Linking Data to Ontologies. *J. Data Semantics* 10 (2008), 133–173.

Provenance-Based Visual Data Exploration with EVLIN

Housseem Ben Lahmar
University of Stuttgart
housseem.ben-lahmar@ipvs.uni-stuttgart.de

Michael Blumenschein
University of Konstanz
michael.blumenschein@uni-konstanz.de

Melanie Herschel
University of Stuttgart
melanie.herschel@ipvs.uni-stuttgart.de

Daniel A. Keim
University of Konstanz
keim@uni-konstanz.de

ABSTRACT

Tools for *visual data exploration* allow users to visually browse through and analyze datasets to possibly reveal interesting information hidden in the data that users are a priori unaware of. Such tools rely on both *query recommendations* to select data to be visualized and *visualization recommendations* for these data to best support users in their visual data exploration process.

EVLIN (exploring visually with lineage) is a system that assists users in visually exploring relational data stored in a data warehouse. EVLIN implements novel techniques for recommending both queries and their result visualization in an integrated and interactive way [3]. Recommendations rely on *provenance* (aka lineage) that describes the production process of *displayed data*.

The demonstration of EVLIN includes an introduction to its features and functionality through sample exploration sessions. Conference attendees will then have the opportunity to gain hands-on experience of provenance-based visual data exploration by performing their own exploration sessions. These sessions will explore real-world data from several domains. While exploration sessions use a Web-based visual interface, the demonstration also features a researcher console, where attendees may have a look behind the scenes to get a more in-depth understanding of the underlying recommendation algorithms.

1 VISUAL DATA EXPLORATION

Data exploration [8] helps users in finding interesting information in data sets when they do not know beforehand what useful information hides in their data. It thus supports humans in understanding and interpreting data in an investigative way. As manual data exploration is tedious, time-consuming, and it is easy to overlook interesting information, there is a need for tools supporting data exploration. These tools typically rely on different kinds of recommendations. Essentially, *query recommendation* guides users in their investigation of a data set D by suggesting queries as next exploration steps, given an initial query Q . Opposed to that, *visualization recommendation* commonly determines suited visualizations given a data set as input.

State-of-the-art. Most data and visualization recommendation techniques work independently from one another, meaning that the result of query recommendation, i.e., the data set $Q'(D)$ returned by executing a recommended query Q' over D , has no impact on the visualization recommendation process, and vice versa. This becomes apparent in Tab. 1 that summarizes works most closely related to ours. For each approach, it describes (i) the expressiveness of input queries (e.g., select-project-join (SPJ) queries, select-project-aggregate (SPA) queries, or cube queries

corresponding to SPJA queries), (ii) the type of recommended output queries, (iii) the information used to compute query recommendations, (iv) the type of recommended visualization, and (v) the information used to compute visualization recommendations. This summary clearly shows that there is a gap between query recommendation systems such as YmalDB [5], SeeDB [12], or REACT [10] for expressive data exploration on the one hand, and visualization recommendation systems such as Voyager [14, 15] and Tableau’s Show Me [9] on the other hand. Indeed, whereas the former may support the full range of typical OLAP queries, they do not offer any visualization recommendation. Typically, there is a one to one mapping between the result relation and a displayed table [5, 10] or bar chart [12]. Opposed to that, visualization recommendation solutions typically offer no or very limited support for query recommendation.

System	input query	recom. query	input for query recom.	recom. vis.	input for vis. recom.
YmalDB [5]	SPJ	SPJ	$D, Q(D)$	-	-
SeeDB [12]	cube	sub-cube	$D, Q(D)$	-	-
REACT [10]	cube	OLAP queries	previous exploration sessions’ query history	-	-
Show Me [9]	SPA	-	-	diverse	data types
Voyager [14, 15]	SPA	Changed SELECT-clause	D, Q , metadata of D (schema, statistics)	diverse	data types, visual encoding channels
EVLIN	cube	OLAP queries	$D, Q, \text{data \& evolution provenance}$	diverse	$Q(D)$, evolution provenance

Table 1: Summary of data exploration systems leveraging query recommendation or visualization recommendation

Contribution. EVLIN bridges the gap between query and visualization recommendation, seamlessly integrating both query and visualization recommendation for a streamlined, interactive user-experience. It relies on a novel recommendation strategy that leverages provenance to recommend queries and interactive visualizations in relation to each other [3]. The underlying techniques as well as the implementation focus on visually exploring relational data stored in a data warehouse. That is, we assume an input data set to conform to a snowflake schema. This demonstration focuses on the usability and interactivity of EVLIN in letting users explore these data. Through various real-world scenarios, we showcase that provenance-based recommendations for visual data exploration allow to effectively reveal interesting information. An example exploration session showing the functionality of EVLIN is available as an online video in [1].

Structure. Sec. 2 highlights the innovative aspects of EVLIN. The audience experience is first addressed in Sec. 3 where we discuss an exploration session in detail. On-site details such as the intended audience, sample scenarios, and a summary of the audience experience beyond the sample exploration session of Sec. 3 are then covered in Sec. 4.

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

2 EVLIN CONTRIBUTIONS

This section briefly summarizes the scientific contributions of EVLIN. We refer interested readers to [3] for more technical details, which we leave out here due to space constraints.

Leveraging data and evolution provenance. EVLIN captures two types of provenance: *data provenance* (more specifically why-provenance) and *evolution provenance* [7]. Data provenance records which data in the database D was used to derive the query result $Q(D)$ of a given query Q . In our context, evolution provenance [3] captures the explored dataset history as well as user interactions and visual encoding parameters of corresponding visualizations, thus tracking how a current visualization was derived. Thus, our model of evolution provenance extends the query history model used for query recommendation in REACT [10].

Provenance-based recommendation. We have developed a novel query recommendation algorithm that takes into account data provenance of data that has been explored and interacted with during an exploration session via the visual front-end. For an input SPJA SQL query, the computed recommendations follow typical data warehouse operations such as drill-down, roll-up, or slice. To identify adequate visualizations for the results of recommended queries, we have further developed a recommendation strategy that takes into account both interactions and visualizations captured as evolution provenance. While our query recommendation is similar in spirit to REACT [10], which records query histories from exploration sessions, our system leverages a richer provenance model and recommends not only queries but also their result visualizations.

Recommendation data space coverage and conciseness. To the best of our knowledge, EVLIN is the first system that recommends visualizations of query results for all queries typical in data warehouse navigation. Indeed, recommendations include roll-up, slice/dice, and drill-down queries, including drill-down queries that navigate to dimensions not considered by previous queries. In addition, data can be clustered by different characteristics or measures to zoom-in to more detailed distributions. To reduce the number of recommendations that are ultimately presented to the user (both for efficiency and usability reasons) while avoiding the loss of potentially relevant recommendations, we have explored how to leverage integrity constraints such as functional dependencies to prune redundant recommendations [3].

Visualization of quantified recommendation quality. Given the high diversity and possibly large number of recommended queries (and associated visualizations) produced by EVLIN, we propose to support users to navigate through the exploration space by quantifying the “interestingness” of recommended next exploration steps. The computed scores are then visualized in an interactive impact matrix, pointing users to potentially interesting data visualizations for different data warehouse operations.

3 SYSTEM FUNCTIONALITY

The above contributions are implemented as part of the EVLIN Web application that supports the exploration of multidimensional relational data stored in a data warehouse D with schema $schema(D)$, given an initial SQL query Q . EVLIN users are expected to have initial basic knowledge about the schema and dimensions of the data warehouse. The query Q , as well as all queries subsequently recommended take the form

`SELECT $f(m)$, A FROM $rel(Q)$ WHERE $cond$ GROUP BY A`

where m is a measure in the fact table, $A = \{a_1, \dots, a_n\}$ is a set of attributes, f is an aggregation function, $rel(Q)$ refers to one or more relations in $schema(D)$, and $cond$ is a conjunction of predicates.

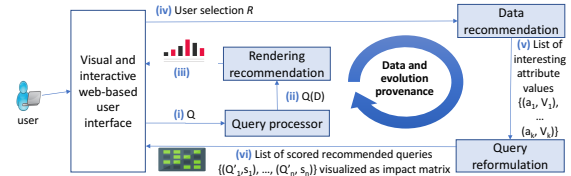


Figure 1: EVLIN system overview

Fig. 1 depicts the general processing EVLIN implements: (i) A user triggers an exploration session by issuing Q . (ii) The *query processor* executes the query and returns its result, denoted $Q(D)$. (iii) $Q(D)$ is then input to *rendering recommendation* that determines an adequate visualization to render the query result. (iv) The user can interact with the data via the graphical user interface, selecting a data-subset of interest, denoted R . (v) Based on this interaction, *data recommendation* identifies which attributes and values within their domain may be of interest to the user at the next step of his data exploration session. (vi) For each data recommendation, *query reformulation* determines variations Q' of the query Q that correspond to different exploration queries over a data cube in a data warehouse (slice, drill-down, roll-up, etc.). The interestingness of these queries is quantified based on the data underlying Q and Q' and the consistency of possible visualizations for $Q'(D)$ wrt evolution provenance. The result of this step is a set of recommended queries and respective visualizations for each recommended attribute of step (v) with associated interestingness scores. This information is visualized as an impact matrix to the user. The user then chooses one particular query to explore next. Upon selection of this query Q' , the next iteration of the exploration process starts.

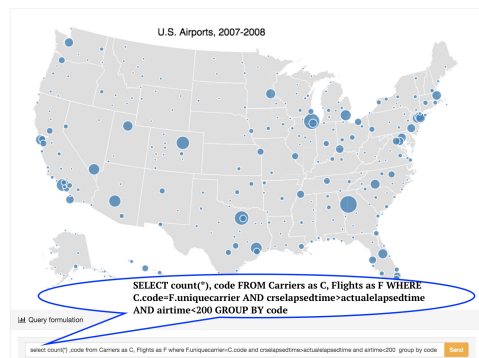
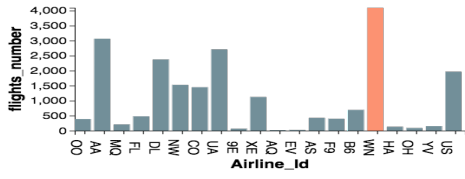


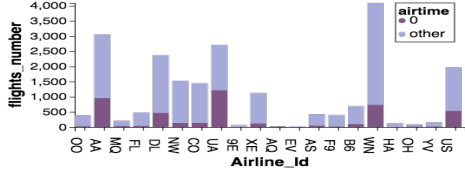
Figure 2: Initial input interface and sample query Q

In what follows, we detail functionalities, interfaces, and interactions that the audience will experience, focusing on the rendering recommendation, data recommendation, and query reformulation components. Due to space constraints, an in-depth discussion of the underlying algorithms is out of the scope of this paper, and we refer interested readers to [3] for details.

An initial user-specified query Q is input via a graphical user interface, as illustrated in Fig. 2. While our current implementation requires full-text SQL (loaded from a file or typed in a text field), a more user-friendly interaction similar to Voyager is planned. Our sample scenario considers a database D of domestic US flights



(a) Example of recommended visualization for $Q(D)$



(b) Visualization of recommended query result $Q'(D)$

Figure 3: Sample visualizations rendered using EVLIN

and Q that determines the number of short flights per airline which arrive ahead of schedule (a possible indicator for airline quality). Using the same scenario, a more extensive exploration session than the one described in the following is showcased in an online video in [1]. It includes for instance visualizations other than bar charts and further data warehouse operations.

3.1 Rendering recommendation

The rendering recommendation component takes as input a query result $Q_i(D)$ and evolution provenance P_e to determine a suited visualization of $Q_i(D)$. As described in [3], evolution provenance encompasses information about (i) past queries, (ii) the set of visualization resources used to render results of these queries, and (iii) information of past user interactions such as the selected data regions or specific sub-results. Essentially, this information are collected for each exploration step of an exploration session, where steps are delimited by transitioning from visualizing a query Q to visualizing a query Q' . This results in evolution provenance being a directed graph where nodes represent individual exploration steps and edges represent transitions from one step to the next. For a given $Q_i(D)$, its corresponding evolution provenance P_e includes all meta-data associated to graph nodes on the path from the initial query Q to the node representing the query Q_i .

Using this input, rendering recommendation aims at maximizing the visual similarity of a recommended visualization with those seen and interacted with previously for similar queries (intuitively, such that users easily recognize the same information as seen previously and thus understand the meaning of visualizations faster). This first requires determining similar queries among those in P_e . We quantify this similarity using a token-based similarity function between Q_i and a query $Q_p \in P_e$, weighted by the inverse of the shortest distance from Q_i to the previously seen Q_p on the path defining P_e . Among all queries of P_e with a similarity to Q_i above a predefined threshold θ_V , we now take the most frequent encoding parameters used to visualize information that is also selected by Q_i . These define a preliminary skeleton for the recommended visualization of $Q_i(D)$. In case no prior visualizations can be used, we resort to the effectiveness metrics adopted by Voyager [14, 15] to construct the visualization. Finally, the visualization skeleton is possibly updated based on the set of constraints defined in [11], that reduce conflict between visualizations.

As an example, Fig. 3a displays the recommended bar-chart visualization for our sample query Q that counts short flights

arriving ahead of schedule for each airline. For this initial rendering, $P_e = \emptyset$, thus, the recommendation is solely based on the effectiveness metrics.

The user can inspect (by mouse hovering) the data and select a region R of interest by clicking to pursue the exploration. In Fig. 3a, the user has selected the highest bar that designates the overly punctual flights of the airline with code WN. The selected region is highlighted in a different color. At this point, P_e is updated to include the initial query Q in its query history, the user interaction event (selection of bar with code=WN) as well as visual encoding parameters of the displayed chart.

Recommending the visualization of Fig. 3b (explained later) relies on the updated P_e to generate a similar visualization of same features, e.g., same axis scale for the y-axis, same order of airline codes on the x-axis, or choice of a stacked bar chart to maintain same heights as seen previously (e.g., in the bar-chart of Fig. 3a).

3.2 Data recommendation

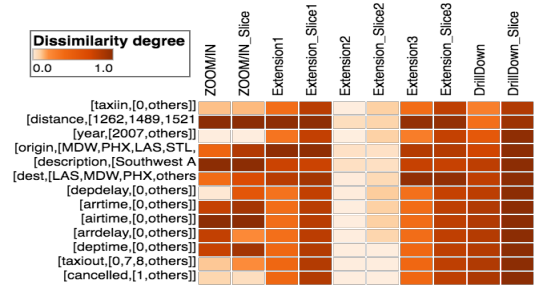


Figure 4: Impact matrix for running example

Selecting a particular sub-result $R \subseteq Q(D)$ of the data via the graphical user interface triggers the data recommendation component. It takes as input R , D , and Q to determine a set of attribute-value pairs $P = \{(a_1, v_1), \dots, (a_n, v_n)\}$. Before passing these to query reformulation, the attribute-value pairs in P are grouped by attribute, resulting in the final output $G = \{(a_1, V_1), \dots, (a_k, V_k)\}$. Essentially, data recommendation determines which data to explore next while query reformulation determines how these data will be explored.

To compute P , we leverage the data provenance of R , denoted $P_d(R)$, using the Perm provenance management system [6]. This provenance corresponds to all tuples in D that have contributed to producing R (i.e., why-provenance [4]). An attribute-value pair is then recommended if it satisfies one of the two following conditions: (i) it is widely present in $P_d(R)$ and more massively present in the database D or (ii) it is widely present in $P_d(R)$ but rarely present in D . We verify these conditions by first requiring a minimum frequency $f_{a,v}$ for an attribute-value pair in $P_d(R)$, i.e., $f_{a,v}(P_d(R)) \geq \theta_L$, where θ_L is a predefined threshold. We then compare this frequency to the frequency of the same attribute-value pair in the whole database D using the support measure defined by $support_{a,v}(R) = \left\lfloor \log_e \left(\frac{f_{a,v}(P_d(R))}{f_{a,v}(D)} \right) \right\rfloor$. Finally, only those attribute-value pairs with a lineage-based support above a given threshold θ_{supp} are retained for recommendation. In our prototype, threshold values of θ_L and θ_{supp} have been set to 0.1 and 0.7 respectively, which proved to be practical for the use cases we considered. However, setting these in general is an interesting avenue for future research. Using the method described above, it is possible that two distinct entries in P , i.e., (a, v) and (a', v') yield redundant query reformulations.

This is for instance the case when functional dependencies exist between attributes. To avoid redundant recommendations, we employ data profiling algorithms to determine functional dependencies [2] of the form $a \rightarrow a'$ and prune a' . The row labels of Fig. 4 show the set G , including for instance $(cancelled, \{1\})$ or $(dest, \{LAS, MDW, PHX\})$ that result from relevant attribute pairs $(cancelled, 1)$ and $\{(dest, LAS), (dest, MDW), (dest, PHX)\} \subseteq P$, respectively.

3.3 Query reformulation

The data recommendations are input to the query reformulation component that produces, for each $(a, V) \in G$, a set of queries corresponding to variations of the original query Q . Each variation reflects an operation typical when querying data warehouses. Our system supports variations implementing slice (and dice), drill-down, including the navigation to dimensions not considered in the initial query Q (which we call extension afterward), roll-up, and grouping or clustering the original results of Q by further attributes (zoom-in). The variations are systematically constructed based on Q , (a, V) , and the specific data warehouse operation. For instance, the query reformulation for slice will add conditions of the form $AND a = v_1 OR \dots OR a = v_n$ to the initial WHERE clause of Q , where $\{v_1, \dots, v_n\} = V$, whereas a roll-up, drill-down changes the attribute set A in the SELECT and GROUP BY clauses of Q to a higher or lower granularity.

To assist users in choosing the next query for the next exploration step, we assign a utility score s to each query variation. Developing and evaluating suited scoring functions to quantify the interestingness of query variations based on their result data and candidate visualization properties is currently actively researched. As a proof-of-concept for our visual data exploration, we have currently implemented Kullback-Leibler divergence function [13] as a utility function. It quantifies the divergence of a 's value distribution in $Q'(D)$ from its distribution in D .

The mapping of (a, V) -pairs to sets of scored recommended queries is visualized as an impact matrix. Fig. 4 shows the impact matrix that results from the query and interaction depicted in Fig. 2 and Fig. 3a. Each line of the matrix corresponds to an (a, V) -pair and each column corresponds to a type of query variation. The cell colors encode the divergence score. From this example, we see for instance that the zoom-in query for $(airtime, 0)$ obtains a high interestingness score. Upon clicking this interesting cell, the corresponding query variation is set to be the new query Q , which is then executed before its result is visualized as recommended by the rendering recommendation component (see Fig. 3b).

The visualization of Fig. 3b shows that a significant number of flights satisfying the initially intended quality criteria for airlines, i.e., arrival ahead of schedule, has a flight duration equal to zero. Based on this insight, users may decide to revise or refine the airline quality criteria.

4 SCENARIOS AND USER EXPERIENCE

The demonstration will rely on scenarios from different domains. The domains are chosen so that some basic knowledge about database schemas and attributes can be assumed. Possible supported scenarios could leverage for instance the following datasets.

Flights. The first dataset describes US domestic flights¹. It contains information about two million flights done by more than 1500 airline companies between 2007 and 2008. It includes further information about 3300 airports and almost 4500 plane types

used for the covered flights. The facts recorded for each flight include various numerical attributes such delays, cancellation, arrival and departure time etc.

Movies. The second dataset describes one million ratings made by 6000 users of the MovieLens platform² on 4000 movies. This database stores various information about users and movies.

Soccer. The third dataset is the European soccer league database³. It contains detailed information about more than 25,000 fixtures between 2008 and 2016 in 11 European championships.

We expect the demonstration to attract a broad audience, generally interested in interactive data analysis or visual data exploration. Having some proficiency in SQL is crucial to be able to follow and express SQL queries that trigger an exploration session.

The audience experience will be similar to the sample scenario used throughout Sec. 3. However, whereas the example limits to one exploration step, attendees will have the opportunity to run exploration sessions spanning multiple exploration steps similarly to the user experience shown in [1].

In addition to experiencing the system's main functionality, the demonstration provides a tour "behind the scenes". This includes seeing how user interaction translates to a provenance query, which scores are computed for data recommendation, which attribute-values are pruned along the way, and which queries (with associated scores) are recommended. Ultimately, after hands-on data exploration experience using EVLIN, the audience will have gained a better understanding of the explored data set and possibly even discovered new insights on the underlying data set.

Acknowledgments. We thank the German Research Foundation (DFG) for supporting projects A03 and D03 of SFB-TRR 161.

REFERENCES

- [1] 2017. EVLIN Demo. (2017). https://youtu.be/L_59cXZu0Uk
- [2] Ziawasch Abedjan, Lukasz Golab, and Felix Naumann. 2015. Profiling relational data: a survey. *VLDB Journal* 24, 4 (8 2015), 557–581.
- [3] Houssein Ben Lahmar and Melanie Herschel. 2017. Provenance-based Recommendations for Visual Data Exploration. In *TaPP*.
- [4] James Cheney, Laura Chiticariu, and Wang Chiew Tan. 2009. Provenance in Databases: Why, How, and Where. *Found. Trends databases* 1, 4 (4 2009), 379–474.
- [5] Marina Drosou and Evaggelia Pitoura. 2013. YmalDB: Exploring relational databases via result-driven recommendations. *VLDB Journal* 22, 6 (12 2013), 849–874.
- [6] Boris Glavic and Gustavo Alonso. 2009. The Perm Provenance Management System in Action. In *SIGMOD*. 1055–1058.
- [7] Melanie Herschel, Ralf Diestelkämper, and Houssein Ben Lahmar. 2017. A Survey on Provenance: What for? What Form? What from? *VLDB Journal* 26, 6 (12 2017), 881–906.
- [8] Stratos Idreos, Olga Papaemmanouil, and Surajit Chaudhuri. 2015. Overview of Data Exploration Techniques. In *SIGMOD*. 277–281.
- [9] Jock Mackinlay, Pat Hanrahan, and Chris Stolte. 2007. Show Me: Automatic Presentation for Visual Analysis. *TVCG* 13 (11 2007), 1137–44.
- [10] Tova Milo and Amit Somech. 2016. REACT: Context-Sensitive Recommendations for Data Analysis. In *SIGMOD*. 2137–2140.
- [11] Zening Qu and Jessica Hullman. 2016. Evaluating Visualization Sets: Tradeoffs Between Local Effectiveness and Global Consistency. In *BELIV*. 44–52.
- [12] Manasi Vartak, Sajjadur Rahman, Samuel Madden, Aditya Parameswaran, and Neoklis Polyzotis. 2015. SEEDB: Efficient Data-Driven Visualization Recommendations to Support Visual Analytics. *Proc. VLDB Endow.* 8, 13 (9 2015), 2182–2193.
- [13] Larry Wasserman. 2013. *All of statistics: a concise course in statistical inference*.
- [14] Kanit Wongsuphasawat, Dominik Moritz, Anushka Anand, Jock Mackinlay, Bill Howe, and Jeffrey Heer. 2016. Voyager: Exploratory Analysis via Faceted Browsing of Visualization Recommendations. *TVCG* 22, 1 (2016), 649–658.
- [15] Kanit Wongsuphasawat, Zening Qu, Dominik Moritz, Riley Chang, Felix Ouk, Anushka Anand, Jock D. Mackinlay, Bill Howe, and Jeffrey Heer. 2017. Voyager 2: Augmenting Visual Analysis with Partial View Specifications. In *CHI*. 2648–2659.

²<http://grouplens.org/datasets/movielens/>

³<https://www.kaggle.com/hugomathien/soccer>

¹<https://stat-computing.org/dataexpo/2009/>

Interactive Visualization of Large Similarity Graphs and Entity Resolution Clusters

Demonstration Paper

M. Ali Rostami, Alieh Saeedi, Eric Peukert, and Erhard Rahm

Database group and ScaDS Dresden/Leipzig, University of Leipzig

{rostami,saeedi,peukert,rahm}@informatik.uni-leipzig.de

ABSTRACT

Entity Resolution (ER) identifies semantically equivalent entities, e.g. describing the same product or customer. It is a crucial and challenging step when integrating heterogeneous (big) data sources. ER approaches typically compute a similarity graph where vertices represent entities and edges (links) connect similar entities. Different clustering algorithms can be applied on such similarity graphs to finally determine groups of matching entities. In this demonstration paper, we introduce a new interactive tool to visualize and thus help to analyze large similarity graphs and large sets of ER clusters. Users can intuitively investigate the link and cluster structure to identify potential problems such as overly large clusters, cluster overlaps or singletons that might indicate the need for repair activities on the ER result. To support large graphs, computation-intensive tasks like layouting and sampling are executed on the server side as parallel or serial processes. The demo walks through different matching and clustering tasks and allows users to interactively explore the results.

1 INTRODUCTION

In the era of big data, one of the challenging data integration tasks is to identify semantically equivalent entities (e.g., describing the same product or customer) in large and heterogeneous data sources. This task is referred to as Entity Resolution (ER) or record linkage [6]. ER typically computes a similarity graph where vertices represent entities and edges (links) connect similar entities with a pair-wise similarity above a predefined threshold. Matching entities can directly be derived from such a similarity graph or from groups of matching entities determined with subsequent clustering algorithms [9]. Our recently introduced framework FAMER (Framework for Multi-source Entity Resolution) supports a parallel computation of such similarity graphs and ER clusters for multiple (≥ 2) data sources [12].

During the continuous development of FAMER, it is difficult to investigate the correctness and efficiency of the certain algorithms and to understand the problems. Such an investigation may lead to introduce better match and cluster algorithms or even an extra postprocessing step of repair (for more details see [13]). However, to identify (or debug) such issues with limited effort and time we see the need for a comprehensive and powerful approach to visually analyze similarity graphs and ER clusterings. Unfortunately, general purpose graph visualization tools like Gephi¹ or Graphviz² have only limited capabilities to

¹<https://gephi.org>

²<http://www.graphviz.org>

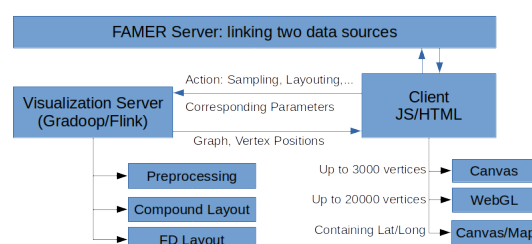


Figure 1: The software architecture

analyze ER clusterings and also have problems to visualize large (big data) similarity graphs with vertex and edge properties.

In this demo paper we introduce SIMG-VIZ, a new visualization system for entity resolution and clustering that allows us to investigate different match and clustering techniques for multi-source entity resolution. SIMG-VIZ offers the following key features:

- SIMG-VIZ allows a user to analyze precomputed similarity graphs and clusterings from existing ER tools and also supports executing and analyzing ER match tasks directly with FAMER.
- Different graph and ER cluster visualization techniques and layouts can be applied to choose the best visualizations.
- To increase performance, some layouts can be precomputed on the server with either parallel or serial computation. This provides a significant optimization potential in particular for force-directed layouts [3].
- To support visualization of large graphs, preprocessing techniques such as sampling (also executed in parallel on the server) can be selected to obtain a fast overview of large similarity graphs and their clustering results.
- Clusters and their overlaps as well as edges annotated with their type and similarity are visualized by using a simple but useful cake-like visual metaphor. Users can interact with clusters and select individual clusters for investigation.

2 SIMG-VIZ OVERVIEW

The SIMG-VIZ system consists of three modules: (1) the FAMER server, (2) a visualization server in JAVA and (3) a web-based UI-client written in JavaScript (see Fig. 1).

The FAMER server is used to link several sources and executes defined matching tasks. However, SIMG-VIZ also allows a user to load similarity graphs and clustering results that were computed by other frameworks and tools. The visualization server offers several preprocessing (e.g. for sampling) and layouting algorithms. The preprocessing algorithms are implemented in distributed fashion based on Gradoop and Apache Flink whereas

graph layouts are currently only implemented as non-distributed algorithms. The web-based client, provides an interactive visualization of similarity graphs and ER-clusterings. Node and edge properties can be investigated and server-side components like matching, clustering, preprocessing and layouting can be triggered. Through the client a user selects options and triggers server based REST-interfaces. The visualization server and the FAMER server both respond with JSON results. In the following paragraphs each of the components is described in detail.

2.1 FAMER Tool

FAMER offers parallel entity resolution for multiple data sources. It supports different match and clustering schemes that are executed on top of a distributed data processing engine (Apache Flink [4]) and the graph analytics framework Gradoop [10]. Its execution has two main phases: (1) computation of a similarity graph based on pairwise matching and (2) clustering. The first phase consists of several steps, namely blocking, pairwise comparisons, and match classification. Blocking reduces the number of necessary comparisons which otherwise would require to compare each entity of a data source to all entities of any other source. Different blocking techniques can be applied in FAMER such as Standard Blocking (SB), Sorted Neighborhood as well as single- and multi-pass blocking. In the matching step all entities of the computed blocks are compared by computing and combining similarities between their attributes, e.g. based on similarity measures such as Edit-Distance or Jaro Winkler. For matching, match rules can specify the required minimal similarity for the considered attribute-comparisons. The output of this step is the set of matching entity pairs (links) together with a combined similarity value per link. In the following match classification step entity pairs are classified as match or no-match based on the computed similarity values. This output is stored as a similarity graph where entities are represented as vertices and match links as edges. The clustering phase of FAMER aims at grouping together all matching vertices of the similarity graph based on the link structure and similarity values. Clustering algorithms typically try to maximize the similarity between entities within a cluster while the similarity between entities of different clusters should be minimized. FAMER supports several clustering algorithms for ER such as Connected Components, Correlation Clustering (CCPivot) [2, 5], Center [9], Merge Center [1], and Star [1].

With SIMG-VIZ, all components of FAMER can be configured and parameterized which allows users to run and compare different ER match tasks.

2.2 Client (Web-based HTML/JS Frontend)

Fig. 2 shows an overview of the web-based client of SIMG-VIZ. At the top part of the client, several options can be selected. The user can choose between different clustering and match configurations which are stored after executing FAMER. Before visualizing the similarity graph some available preprocessing algorithms such as sampling can be applied. Moreover, the user can select layouting options, i.e. which layout to use and where the computation should take place. In particular for large graphs, computing the layout on the server gives significant improvements. Finally, SIMG-VIZ offers a list of actions (see Table 1) that support different drawing tasks of the graph and some statistics computations can be triggered. On the right side of the client a number of parameters for visualization can be set. To improve interactivity,

styling tasks are performed on the client, for example changing vertex or edge sizes.

Table 1: Actions in SIMG-VIZ

Action	Description
Draw graph (Cytoscape)	Draws a sim-graph in Cytoscape ³ .
Draw graph (WebGL)	Draws a sim-graph in WebGL based on VivaGraph.
Compute only	Executes preprocessing and sampling without visualization.
Compute labels/keys	Computes all labels and property-keys of the vertices and edges for filtering in the left part of the UI.
Save as image	Exports an image of the drawn graph.
Remove selected node	Removes a selected node.
Degree Distribution	Computes the degree distribution of the graph.
Graph Statistics	Computes additional basic statistics of a graph.

2.3 Visualization Server

The visualization server offers preprocessing and layouting services. The preprocessing algorithms are implemented in a distributed fashion based on Gradoop and Apache Flink. Table 2 lists currently implemented preprocessing components. All layouting

Table 2: Preprocessing algorithms in SIMG-VIZ

Preprocessing	Description
Graph sampling	Computes a statistical sampling of a graph. Currently SIMG-VIZ implements vertex, edge and page rank sampling.
Graph summary	Computes a graph summary by grouping dense subgraphs of a graph to generate a compact overview of a large graph [11]. In SIMG-VIZ the Flink implementation is used.
Cluster neighbor filtering	In particular for ER-Clustering a filtering to neighbors of clusters is needed. The user specifies a cluster ID (at the right part of UI) and that cluster together with its neighbor clusters is visualized.
Cluster sizes filtering	Only the clusters with specific sizes are visualized.
Cluster Aggregation	It visualizes a graph in which the cluster vertices are grouped together.

algorithms are available both for the client and the server as non-distributed algorithms. We observed that executing layouting algorithms that need iterations like the force directed layout [8] should not be executed on the client within a browser. Executing it on a server brings significant run-time improvements, even without distributing the computation to multiple nodes. After a layout computation finishes on the server, the positions of nodes are send to the client together with the graph. As future work we plan to implement parallel versions of layouting to be run on top of Flink or Gradoop.

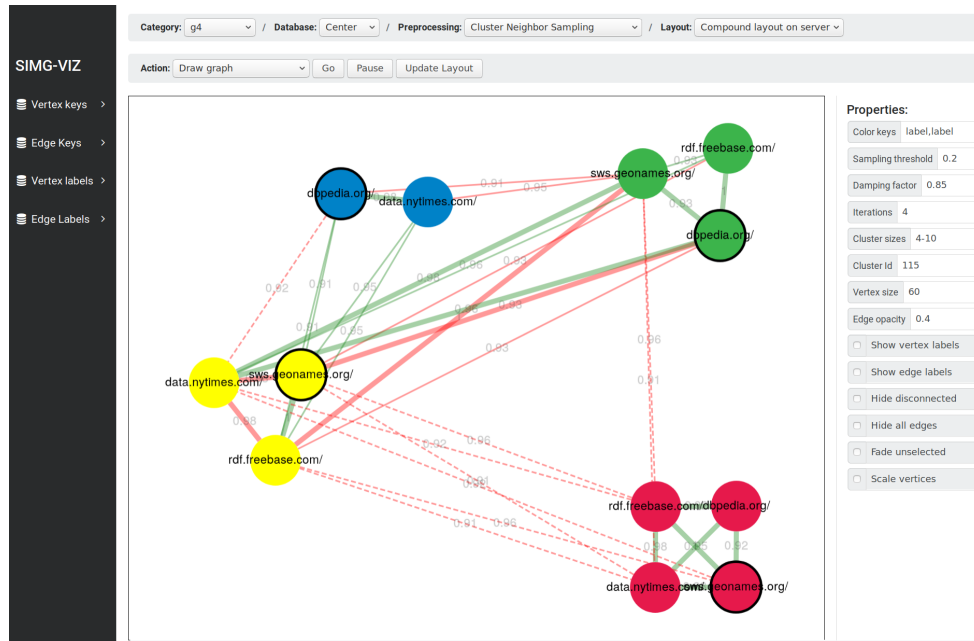


Figure 2: An overview of SIMG-VIZ

2.4 Cluster Visualization

In this section, we describe some specific features of SIMG-VIZ which are designed particularly for the visualization of ER clusters.

These features are explained along a real world ER-example of integrating four duplicate-free data sources namely Freebase (Fb), New York Times (nyt), DBpedia (db), and Geonames (geo). We initially compute a similarity graph and apply different clustering techniques with FAMER. A result cluster includes only vertices from these four sources which are most probably the same real world entities. An edge connects two vertices which have a high value of computed similarity measure. In Fig. 3 we initially visualize the complete clustering result. Clusters are given different colors to indicate cluster membership. Users can identify clusters that may warrant a closer inspection, e.g., clusters with more than 4 vertices or singleton clusters. A user can zoom in and inspect the properties of each vertex. Since generic graph layouting algorithms often have problems in visualizing large similarity graphs (e.g., problem of edge cluttering) we applied a compound layout for cluster visualization. Such compound layouts of graphs like CoSE-Bilkent [7] visualize vertices in a cluster (referred to as compound in the paper) close to each other while the whole graph is visualized by using a modified force-directed algorithm. Fig. 4(middle) shows a visualization of such a compound layout which we also compute on the server side. Vertices of a cluster share the same color and are closely grouped together to form a compound.

To get a cleaner picture, a user can interactively select a specific cluster or enter a cluster ID to only visualize a specific cluster for closer inspection. Often we also need to visualize a cluster together with its neighboring clusters (see Fig. 4 (top)). The vertex labels here refer to the corresponding data source for that entity. The scenario illustrates a problem case since there are more than four cluster members with some data sources having two entities in the same cluster which should not be possible for duplicate-free sources. There is also a singleton cluster that might have

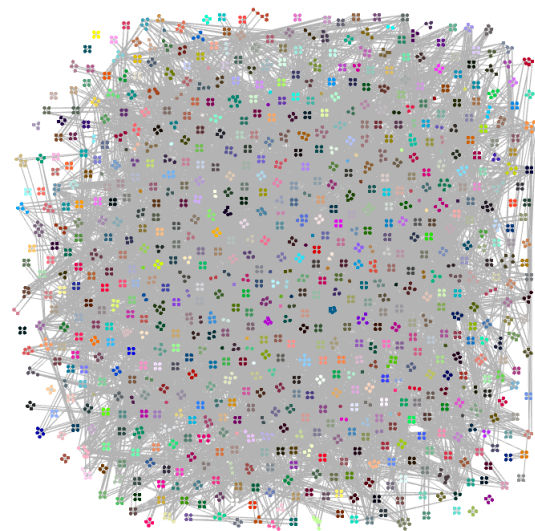


Figure 3: A visualization of all clusters.

to be merged with another cluster. Based on such observations we are now able to re-assess the used cluster algorithms and investigate new approaches for cluster repair.

We also provide support for visualizing clustering result of specific clustering algorithms like Star[9]. The Star clustering computes cluster representatives and all neighbors of those representatives are assigned to the corresponding cluster. In SIMG-VIZ those cluster representatives are highlighted with a black outline (see Figure 4). It happens that a vertex is a neighbor of several cluster representatives so that such vertex will belong to multiple clusters. These multi-assignments are represented as pie-charts on nodes. Each piece of a pie chart which has a specific color specifies a cluster assignment. For example, Fig. 4 (Middle) contains a

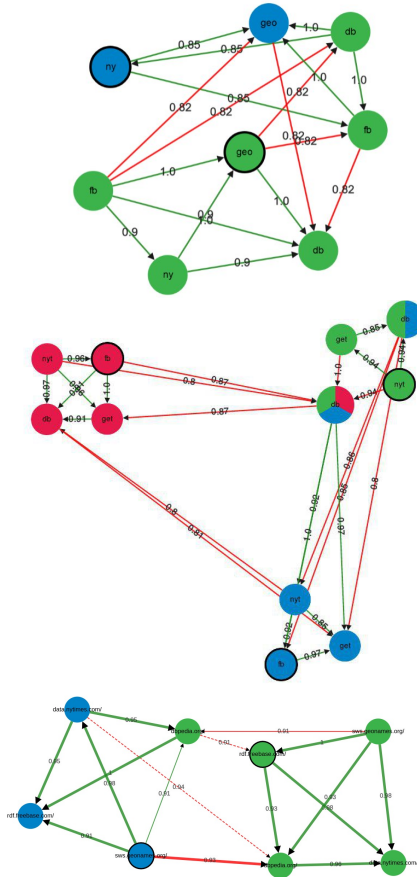


Figure 4: visualizations of (top) clusters with different colors, (Middle) vertices can belong to more than one cluster - drawn in compound layout, (Bottom) different styles for edges indicating how strong connections between entities are.

pie chart with three pieces which means that node (entity) is assigned to three clusters. Obviously some cluster post-processing is needed to select the best cluster for each entity that have been assigned to several clusters.

SIMG-VIZ provides special visualization support for evaluating clusters when the perfect cluster result is available for comparison. As shown in Fig. 4 (Bottom) there are different edge colors: green for edges within correct clusters and red for (wrong) edges between such clusters. Hence, clusters containing red edges should be investigated more closely. Finally we implemented a map-visualization of geo-referenced data so that entities can be plotted onto a map. With the help of this map-visualization we are able to identify false matches within a given data set.

2.5 Web-based Visualization Libraries

Drawing large graphs within a browser is problematic. We investigated several different Javascript-based visualization libraries and observed that there are significant performance differences. Three groups of libraries can be found: (1) SVG-based libraries compute SVG-nodes and tags. They are often feature rich but do not scale well due to many generated SVG-Elements. (2) The second group relies on HTML-Canvas. These libraries are typically

faster but interactivity is harder to realize. Still they mostly do not scale well for larger graphs. (3) WebGL-based libraries offer the best scalability but are still not as feature rich as existing libraries in the other two categories. We finally decided to use the Canvas-based Cytoscape⁴ library for small graphs up to 2000 vertices since it gives more flexibility regarding the style of edges and vertices. For example, the feature of drawing a pie chart on vertices is already available in Cytoscape. For large graphs, we use VivaGraph⁵, which is a WebGL-based library with less support for vertex and edge attributes, styling and coloring. However for larger graphs the user won't be able to see those details anyway.

3 DEMONSTRATION

In the demonstration, we walk through a complete workflow of entity resolution using matching and clustering for small and large ER match problems. A user could select data sources and the corresponding properties, similarity measures and match classifiers that are used by FAMER. The resulting clusters from FAMER are loaded into the visualization server. We then allow a user to compare different preprocessing algorithms as well as different layouts. We consider small data sources as well as large ones.

ACKNOWLEDGEMENT

This work was partly funded by the German Federal Ministry of Education and Research within the projects Competence Center for Scalable Data Services and Solutions (ScaDS) Dresden/Leipzig (BMBF 01IS14014B) and BIGGR (BMBF 01IS16030B).

REFERENCES

- [1] J.A. Aslam, E. Pelekho, and D. Rus. 2006. *The Star Clustering Algorithm for Information Organization*. Springer.
- [2] Ni. Bansal, A. Blum, and S. Chawla. 2004. Correlation Clustering. *Machine Learning* 56, 1 (2004), 89–113.
- [3] G.D. Battista, P. Eades, R. Tamassia, and I.G. Tollis. 1998. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall.
- [4] P. Carbone, . Katsifodimos, S. Ewen, V. Markl, S.Haridi, and Ko. Tzoumas. 2015. Apache Flink: Stream and Batch Processing in a Single Engine. *IEEE Data Engineering Bulletin Issues* 38, 4 (2015).
- [5] F. Chierichetti, N. Dalvi, and R. Kumar. 2014. Correlation Clustering in MapReduce. In *Proc. 20th ACM SIGKDD*. 641–650.
- [6] P. Christen. 2012. *Data Matching: Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection*. Springer.
- [7] U. Dogrusoz, E. Giral, A. Cetintas, A. Civril, and E. Demir. 2009. A layout algorithm for undirected compound graphs. *Information Sciences* 179, 7 (2009), 980–994.
- [8] Thomas M. J. Fruchterman and Edward M. Reingold. 1991. Graph Drawing by Force-directed Placement. *Softw. Pract. Exper.* 21, 11 (Nov. 1991), 1129–1164. <https://doi.org/10.1002/spe.4380211102>
- [9] O. Hassanzadeh, F. Chiang, R.J. Miller, and H.C. Lee. 2009. Framework for Evaluating Clustering Algorithms in Duplicate Detection. *PVLDB* 2, 1 (2009), 1282–1293.
- [10] M. Junghanns, A. Petermann, N. Teichmann, K. Gómez, and E. Rahm. 2016. Analyzing Extended Property Graphs with Apache Flink. In *Proc. SIGMOD Workshop on Network Data Analytics*.
- [11] M. Junghanns, A. Petermann, N. Teichmann, and E. Rahm. 2017. The Big Picture: Understanding large-scale graphs using Graph Grouping with GRADOOP. In *Proc. BTW*.
- [12] A. Saeedi, E. Peukert, and E. Rahm. 2017. *Comparative Evaluation of Distributed Clustering Schemes for Multi-source Entity Resolution*. Springer LNCS, 278–293.
- [13] A. Saeedi, E. Peukert, and E. Rahm. 2018. Using Link Features for Entity Clustering in Knowledge Graphs. In *Extended Semantic Web Conference*. submitted for publication.

⁴<http://js.cytoscape.org>

⁵<https://github.com/anvaka/VivaGraphJS>

FastOFD: Contextual Data Cleaning with Ontology Functional Dependencies

Zheng Zheng
McMaster University
zhengz13@mcmaster.ca

Morteza Alipour Langouri
McMaster University
alipoum@mcmaster.ca

Zhi Qu, Ian Currie
McMaster University
{quz1, currie}@mcmaster.ca

Fei Chiang
McMaster University
fchiang@mcmaster.ca

Lukasz Golab
University of Waterloo
lgolab@uwaterloo.ca

Jaroslav Szlichta
University of Ontario IT
jaroslav.szlichta@uoit.ca

ABSTRACT

Functional Dependencies (FDs) define attribute relationships based on syntactic equality. As a result, FD-based data cleaning systems incorrectly label syntactically different but semantically equivalent values as errors. To address this problem, we demonstrate FastOFD: a system for discovering Ontology Functional Dependencies (OFDs) which express semantic attribute relationships such as synonyms and is-a hierarchies defined by an ontology. In addition to discovering OFDs, FastOFD generates suggestions for repairing erroneous data, and we demonstrate that FastOFD significantly reduces the number of false positive errors compared to FD-based data cleaning techniques.

1 INTRODUCTION

In constraint-based data cleaning, tuples that violate the given integrity constraints are identified as erroneous. Candidate repairs are then generated to suggest how erroneous tuples could be modified to eliminate inconsistencies. The most popular integrity constraint considered in the data cleaning literature has been the Functional Dependency (FD) [2] and its extensions such as conditional FDs [3]. However, FDs are limited to identifying attribute relationships based on syntactic equivalence or syntactic similarity in case of metric FDs [5]. As a result, data cleaning systems that are based on FDs have a common flaw: they incorrectly label syntactically different but semantically equivalent values as errors. This leads to an increased number of reported “errors” and a larger search space of candidate data repairs.

Example: Table 1 shows a sample of clinical records containing patient country codes (CC), country (CTRY), symptoms (SYMP), diagnosis (DIAG), and the prescribed medication (MED). Consider two FDs: $F_1: [CC] \rightarrow [CTRY]$ and $F_2: [SYMP, DIAG] \rightarrow [MED]$. The tuples (t_1, t_5, t_6) violate F_1 as ‘United States’, ‘America’, and ‘USA’ are *not syntactically equivalent* (the same is true for (t_2, t_4, t_7)). However, ‘United States’ is synonymous with ‘America’ and ‘USA’, and (t_1, t_5, t_6) all refer to the same country. Similarly, ‘Bharat’ in t_4 is synonymous with ‘India’ as it is the country’s original Sanskrit name. For F_2 , (t_1, t_2, t_3) and (t_4, t_5, t_6) do not satisfy the dependency as the consequent values all refer to different medications. However, with domain knowledge from a medical ontology shown in Figure 2, we see that the values participate in an inheritance relationship. Both ‘ibuprofen’ and ‘naproxen’ are non-steroidal anti-inflammatory drugs (NSAID), and ‘tylenol’ is an ‘acetaminophen’ drug, which in turn is an ‘analgesic’.

id	CC	CTRY	SYMP	DIAG	MED
t_1	US	United States	joint pain	osteoarthritis	ibuprofen
t_2	IN	India	joint pain	osteoarthritis	NSAID
t_3	CA	Canada	joint pain	osteoarthritis	naproxen
t_4	IN	Bharat	nausea	migrane	analgesic
t_5	US	America	nausea	migrane	tylenol
t_6	US	USA	nausea	migrane	acetaminophen
t_7	IN	India	chest pain	hypertension	morphine

Table 1: Sample clinical trials data

To address these problems, we demonstrate FastOFD¹, a tool for *contextual* data cleaning with a novel class of dependencies called Ontology Functional Dependencies (OFDs) which take synonyms and inheritance relationships into account. In the above example, if F_1 and F_2 were specified as OFDs then no tuples would be falsely reported as erroneous. Our demonstration focuses on the following novel features of FastOFD:

- (1) **Automatic discovery of OFDs to show how prevalent they are in real data.** We have recently proposed an efficient algorithm for discovering OFDs from data [1]. The FastOFD system uses this algorithm, and conference participants will be able to run it on several real datasets and ontologies, and visualize the results. Furthermore, conference participants will learn, through real examples, about an interesting aspect of OFDs that does not arise in standard FDs: the notion of senses or interpretations with respect to a given ontology. For example, the value “jaguar” can be interpreted as an animal or as a vehicle.
- (2) **Data cleaning with OFDs.** FastOFD discovers OFDs that mostly hold but may be violated by a bounded number of tuples. Once such OFDs are discovered, FastOFD identifies erroneous tuples and suggests how to modify them in order to remove inconsistencies. Conference participants will be able to apply the suggested modifications (or propose different modifications) in real-time.
- (3) **Comparison with FD-based data cleaning methods.** We demonstrate that FastOFD is practically as fast as existing algorithms for discovering traditional FDs, yet OFDs are more expressive. We also show that FastOFD significantly reduces the number of false positive errors compared to FD-based data cleaning techniques. This reduces the computational effort of data cleaning and the manual burden for users to identify falsely categorized data errors.

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

¹<http://db.cas.mcmaster.ca/fastofd>

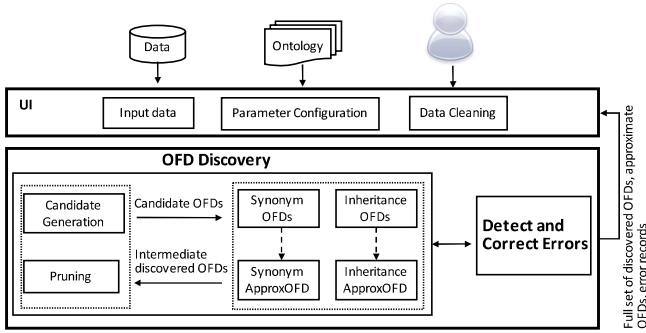


Figure 1: FastOFD Architecture.

2 SYSTEM OVERVIEW

2.1 Ontology Functional Dependencies

A functional dependency (FD) F over a relation R is represented as $X \rightarrow A$, where X is a set of attributes and A is a single attribute in R . An instance I of R satisfies F if for every pair of tuples $t_1, t_2 \in I$, if $t_1[X] = t_2[X]$, then $t_1[A] = t_2[A]$. A partition of X , Π_X , is the set of equivalence classes containing tuples with equal values in X . For example, in Table 1, $\Pi_{CC} = \{\{t_1, t_5, t_6\}\{t_2, t_4, t_7\}\{t_3\}\}$.

An ontology S is a specification of a domain that includes concepts, entities, properties, and relationships among them. The meaning of these constructs can be modeled according to different senses leading to different ontological interpretations.

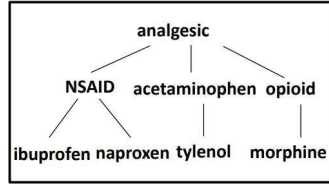


Figure 2: A medical ontology

For example, the value ‘jaguar’ can be interpreted as an animal or as a vehicle. As an animal, ‘jaguar’ is synonymous with ‘panthera onca’, but not with the value ‘jaguar land rover’ which is an automotive company.

We define classes E to capture the senses defined in S . Let $synonyms(E)$ be the set of all synonyms for a class E . For instance, $synonyms(E1) = \{\text{‘car’}, \text{‘auto’}, \text{‘vehicle’}\}$, $synonyms(E2) = \{\text{‘jaguar’}, \text{‘jaguar land rover’}\}$ and $synonyms(E3) = \{\text{‘jaguar’}, \text{‘panthera onca’}\}$. Let $names(C)$ be the set of all classes, i.e., interpretations or senses, for a given value C . For example, $names(jaguar) = \{E2, E3\}$ as jaguar can be an animal or a vehicle. Let $descendants(E)$ be a set of all string representations for the class E or any of its descendants, i.e., $descendants(E) = \{s \mid s \in synonyms(E) \text{ or } s \in synonyms(E_i), \text{ where } E_i \text{ is-a } E_{i-1}, \dots, E_1 \text{ is-a } E\}$. For instance, $descendants(E3) = \{\text{‘jaguar’}, \text{‘peruvian jaguar’}, \text{‘mexican jaguar’}\}$.

We consider two ontological relationships in the right-hand-side of a dependency, synonyms and inheritance, leading to the following definitions for synonym OFDs and inheritance OFDs:

Definition 2.1. A relation instance I satisfies a *synonym OFD* $X \rightarrow_{syn} A$, if for each equivalence class $x \in \Pi_X(I)$, there exists an interpretation in S under which all the A -values of tuples in x are synonyms. That is, $X \rightarrow_{syn} A$ holds if for each class x ,

$$\bigcap_{names(a), a \in \{t[A] \mid t \in x\}} \neq \emptyset.$$

Definition 2.2. Let θ be a threshold representing the allowed path length between two nodes in S over an attribute A (each attribute can have a different θ). A relation instance I satisfies an *inheritance OFD* $X \rightarrow_{inh} A$ if for each equivalence class $x \in \Pi_X(I)$, the A -values of all tuples in x are descendants of

a Least Common Ancestor (LCA) which is within a distance of θ in S . That is, $X \rightarrow_{inh} A$ holds if for each equivalence class x ,

$$\bigcap_{descendants(names(a), a \in \{t[A] \mid t \in x\})} \neq \emptyset$$

and the resulting LCA is within a distance of θ in S to each a .

Example: Consider the OFD $\phi_1: [CC] \rightarrow_{syn} [CTRY]$ from Table 1. We have $\Pi_{CC} = \{\{t_1, t_5, t_6\}\{t_2, t_4, t_7\}\{t_3\}\}$. The first equivalence class, $\{t_1, t_5, t_6\}$, representing the value ‘US’, corresponds to three distinct values of CTRY. According to a geographical ontology, $names(\text{‘United States’}) \cap names(\text{‘America’}) \cap names(\text{‘USA’}) = \text{‘United States of America’}$. Similarly, the second class $\{t_2, t_4, t_7\}$ gives names(‘India’) \cap names(‘Bharat’) = ‘India’. The last equivalence class $\{t_3\}$ contains a single tuple, so there is no conflict. Since all references to CTRY in each equivalence class resolve to some common interpretation, ϕ_1 holds over Table 1. Now consider the OFD $\phi_2: [SYMP, DIAG] \rightarrow_{inh} [MED]$, and the ontology in Figure 2. For the first equivalence class, $\{t_1, t_2, t_3\}$, the LCA is ‘NSAID’ which is within a distance of one to each MED value in this class. For the second equivalence class, $\{t_4, t_5, t_6\}$, the LCA is ‘analgesic’, which is within a distance of two to each MED value in this class. The third equivalence class consists of a single tuple (t_7) so there is no conflict. Thus, ϕ_2 holds with $\theta = 2$ over MED.

Synonym OFDs subsume traditional FDs, where all values are assumed to have a single canonical name (i.e., for all classes E , $|synonyms(E)| = 1$). Furthermore, inheritance OFDs subsume synonym OFDs since we can recover synonym OFDs by setting $\theta = 0$ for each attribute. For example, the inheritance OFD $[SYMP, DIAG] \rightarrow_{inh} [MED]$ allows synonyms such as ‘ibuprofen’ and ‘advil’ to appear in tuples having the same SYMP and DIAG values; however, different medications under the same LCA such as ‘ibuprofen’ and ‘naproxen’ are not allowed.

OFDs *cannot* be reduced to traditional FDs or Metric FDs [5] (which assert that two tuples whose left-hand side attribute values are equal must have syntactically similar righthand side attribute values according to some distance metric). Since values may have multiple senses (e.g., jaguar the animal and jaguar the car), it is not possible to create a normalized relation instance by replacing each value with a unique canonical name. Furthermore, ontological similarity is not a metric since it does not satisfy the identity of indiscernibles (e.g., for synonyms).

2.2 OFD Discovery

The notion of senses makes OFDs non-trivial and has important implications on their discovery. While checking pairs of tuples is sufficient to identify violations and therefore verify traditional FDs (or metric FDs), this is not the case for OFDs. To verify an OFD, we must find a common interpretation of the Y -values for each equivalence class, as shown in Definitions 2.1 and 2.2. As a result, existing dependency discovery algorithms that validate dependencies via pairwise tuple comparisons (e.g., FastFD [6]) cannot be easily extended to OFDs.

Instead, we use our OFD discovery algorithm [1] which uses an Apriori-like approach, similar to the TANE FD discovery algorithm [4]. In this approach, the set of possible antecedent and consequent values considered by FastOFD is modeled as a set containment lattice. OFD candidates are considered by traversing the lattice in a breadth-first search manner. We consider all X consisting of single attribute sets, followed by all 2-attribute sets, and continue level by level to multi-attribute sets until (potentially) level $k = n$, where n is the number of attributes. When the

algorithm processes an attribute set X , it verifies candidate OFDs of the form $(X \setminus A) \rightarrow A$, where $A \in X$. This guarantees that only non-trivial OFDs are considered. For each candidate, we check if it is a valid synonym or inheritance OFD as per Definition 2.1 and Definition 2.2. This small-to-large search strategy guarantees that only minimal OFDs are discovered, and we develop novel optimizations to prune the search space effectively [1]. In our evaluation, we found that FastOFD incurred an average overhead of 5% to 10% over TANE [4] while discovering a larger set of (synonym, inheritance and approximate) dependencies.

2.3 Data Cleaning with Approximate OFDs

In addition to OFDs that hold over the entire data instance, FastOFD can discover approximate OFDs that hold with some exceptions. We define a minimum support level, τ , that specifies the minimum number of tuples that must satisfy an OFD ϕ . Given a relational instance I , and a minimum support threshold τ , $0 \leq \tau \leq 1$, FastOFD can find all minimal OFDs ϕ such that $s(\phi) \geq \tau$ where $s(\phi) = \max\{|r| \mid r \subseteq I, r \models \phi\}$. Since approximate OFDs are satisfied by most of the relation, violating tuples may be considered erroneous. These are records involving syntactically different but semantically equivalent attribute values that would otherwise be identified as errors using traditional FDs.

Furthermore, unlike previous dependency discovery algorithms, FastOFD discovers OFDs that hold under a given sense, which provides context for data cleaning. For a given ontology, we assume a sense is provided that defines the ontological context under which we identify the exceptions and the corresponding fixes (based on satisfying record values). For synonym OFDs, the set of potential clean values includes all synonyms defined in the ontology. For inheritance OFDs, we identify the highest ancestor a of the frequent (clean) values in the data, and record the set of ontology values from the sub-tree with a as the root. We ensure that the height of the sub-tree does not exceed θ .

2.4 FastOFD Architecture

FastOFD is an interactive web application running on Flask + uWSGI using Python and JavaScript libraries. The backend is implemented using Java v1.8, running on an Intel Xeon CPU E7-LL8867 2.13GHz with 32GB of memory. Figure 1 shows the FastOFD architecture, consisting of a user-interface (UI) layer, the OFD discovery engine, and a data cleaning module that identifies and corrects error values by interactively engaging with a user. The user interface (UI) layer provides the input specifications for the data instance, parameter configuration, and ontology RDF files along with the corresponding attributes and senses. Configuration settings include specifying the path length threshold (θ), the desired type of OFDs to be discovered (synonym or inheritance, or both), whether approximate OFDs are desired, and if so, the minimum support level. Discovered OFDs are returned to the user along with approximate OFDs for interactive cleaning.

The discovery algorithm generates candidate OFDs by traversing the attribute lattice in a level-wise manner. FastOFD checks whether a candidate $\phi : X \rightarrow A$ satisfies a synonym or inheritance OFD relative to the given parameter settings. If so, then candidate OFDs that are supersets of X are pruned from the lattice, and ϕ is added to the list of discovered OFDs. If ϕ is an approximate OFD (i.e., with support greater than τ), we record the satisfying (clean) values along with the exceptions, and continue the search by evaluating candidate OFDs containing supersets of X . The efficiency of our algorithm relies on pruning candidates

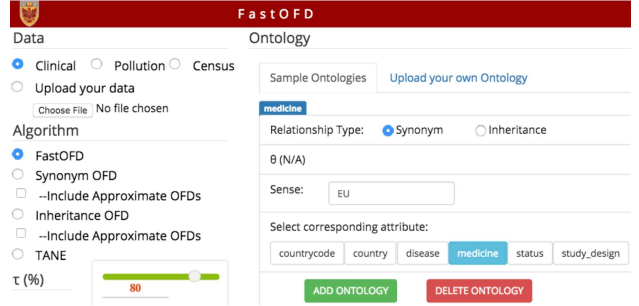


Figure 3: Input and parameter configurations.

through our axioms from the lattice that are supersets of discovered OFDs, keys, and FDs. We also disqualify approximate OFD candidates that do not have a minimum τ support. That is, for a candidate $X \rightarrow A$, adding a new attribute B to X only fragments the equivalence classes, and does not increase the support level.

3 DEMONSTRATION OVERVIEW

In this demo, conference participants will interact with FastOFD to: (1) discover OFDs and see first-hand how prevalent they are in real data. We show examples from both synonym and inheritance OFDs, and cases where similar OFDs exist under different senses; (2) interactively clean the data using approximate OFDs where candidate repairs are provided, and users can apply these changes in real-time with immediate feedback on the impact to support levels; and (3) experience the effectiveness of FastOFD to reduce the number of false positive errors while achieving comparable runtimes to traditional FD discovery algorithms.

Input Interface. Figure 3 shows the input interface where users can select a sample dataset among clinical trials data from LinkedCT.org, pollution data from the Canadian Pollutant Release Inventory, and census-income data from <https://archive.ics.uci.edu/ml/>. In this paper, we use the clinical data to demonstrate example scenarios. Users can then select the type of dependencies, configure τ to return approximate OFDs with minimum τ support, and inheritance OFDs containing IS-A relationships within a distance of θ in the ontology. For each input ontology, users can define the corresponding synonym or inheritance relationship, the applicable attribute, and the sense interpretation. For example, in Figure 3, we add an ontology containing synonyms for attribute ‘medicine’ to be interpreted under the sense ‘EU’ for European Union.

3.1 Automatic Discovery of OFDs

Figure 4 shows an example output of two discovered synonym OFDs, and four approximate synonym OFDs ranked according to decreasing support. Each dependency shows the corresponding sense under which it holds over the data. The inclusion of senses is unique to FastOFD (in contrast to other dependency discovery systems) as it provides a specific interpretation under which an OFD holds. For example, Figure 4 shows that the synonym OFD [disease] \rightarrow [medicine] holds over the clinical data under two senses: (i) as an OFD under sense ‘EU’ (for the ‘European Union’), and (ii) as an approximate (synonym) OFD under sense ‘US’ (for the ‘United States’) with 93.4% support. This example shows that the same medication may be used to treat different diseases, and is referred to by different names, in different regions of the world (i.e., in different geographical senses). Senses help to differentiate semantically equivalent entities under different contexts. Similarly, the OFD [country] \rightarrow [countryCode] holds

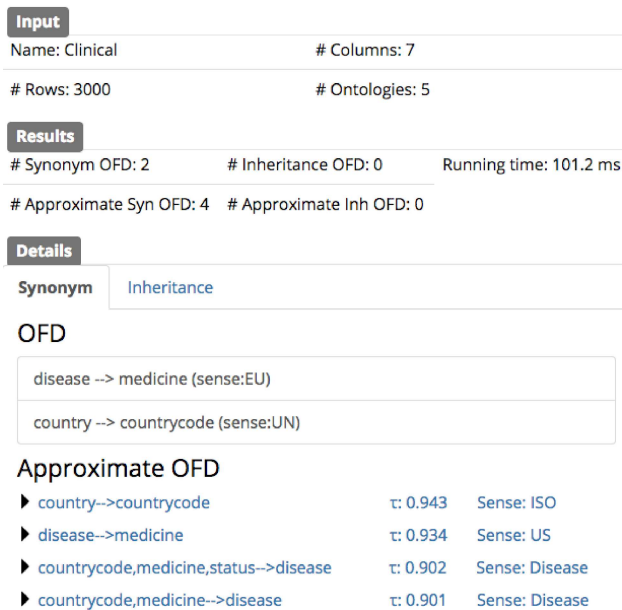


Figure 4: Viewing discovered OFDs.

under the sense ‘UN’ denoting country name abbreviations published by the United Nations, but holds approximately under the sense ‘ISO’ denoting the abbreviations used by the International Organization for Standardization. Conference participants will be able to delve deeper (by clicking on each OFD) to explore these distinctions between similar dependencies. Unlike previous dependency discovery algorithms that assume equivalence classes, FastOFD provides a contextual view for discovered OFDs that can be customized with interpretations according to a user’s domain or application requirements.

3.2 Contextual Data Cleaning

Given a set of approximate OFDs, we now show how users can interact with FastOFD to correct these exceptions. Figure 5 shows a data cleaning example w.r.t. the OFD ϕ : [disease] \rightarrow [medicine], that states a given disease is treated by prescribed medications. However, while ϕ holds over the clinical data under the sense ‘EU’, it holds only approximately under the ‘US’ sense as medication names vary between the two regions. An example of this semantic data quality issue is shown in Figure 5 by expanding the approximate OFD version of ϕ . Users can view the positive (supporting) examples highlighted in green, while the exceptions are highlighted in red. In this example, ‘asthma’ is treated by medications {‘Advicor’, ‘Advair’, ‘Seretide’}, which are all synonymous in the ‘EU’ sense. However, in the ‘US’, ‘Seretide’ is not recognized as a synonym, and is flagged as an exception. The frequency levels for each tuple pattern are shown such that users can correct exceptions directly, and receive immediate feedback. By clicking ‘Apply’, if the entered value(s) are correct, the corresponding τ support level for the OFD is updated, and the tuple pattern changes from red to green. Conference participants will be able to perform hands-on exploration and cleaning, and observe how prevalent these OFDs are in real data.

3.3 Effectiveness of FastOFD

Finally, we show that FastOFD significantly reduces the number of ‘false positive’ errors that are found in FD-based data cleaning solutions, with comparable running times to FD discovery algorithms even though OFDs have greater expressiveness and

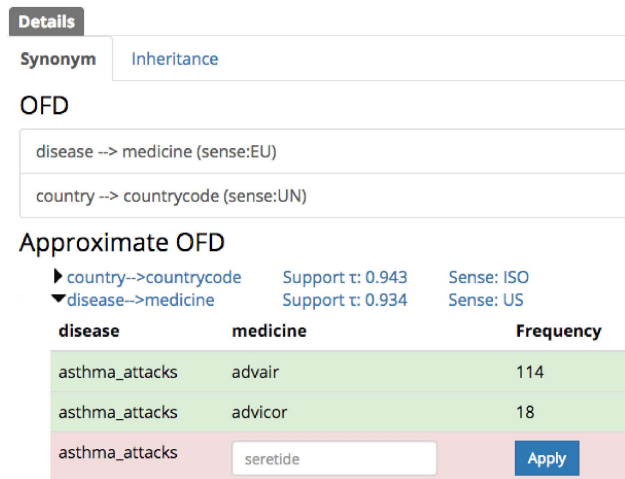


Figure 5: Data cleaning with approximate OFDs.

larger result sets [1]. Figure 6 shows the comparative performance between FastOFD and TANE, and more importantly, the distinctions between the two sets of approximate dependencies. While the same set of dependencies are discovered, the support level τ is higher for approximate OFDs than approximate FDs. Upon closer inspection, users will be able to drill down and notice that these differences are caused not only due to true errors (highlighted in red), but also the false positive errors (highlighted in blue). These false positive values represent values that are *syntactically* different than the clean (green highlighted) values, but are *semantically* equivalent. For example, the medication ‘Celexa’ is synonymous with ‘Celebrex’ in treating ‘osteoarthritis’. These values are identified as clean in FastOFD but as exceptions in TANE. Our case studies show that an average 33% of errors found in traditional FD-based data cleaning solutions are false positive errors that can be eliminated. This reduces the manual burden of having users validate these ‘errors’, and also reduces the space of possible repairs, thereby improving the overall data cleaning runtime. In addition to the synonym examples described here, conference participants will be able to explore examples involving inheritance properties that are present in real data.

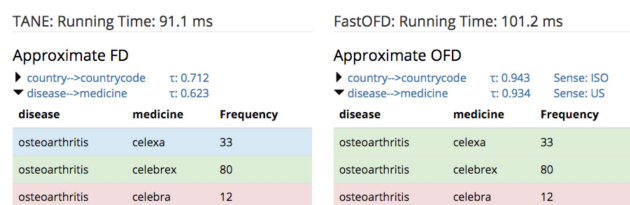


Figure 6: Reducing the number of false positives.

REFERENCES

- [1] S. Baskaran, A. Keller, F. Chiang, L. Golab, and J. Szlichta. Efficient discovery of ontology functional dependencies. In *CIKM*, pages 1847–1856, 2017.
- [2] P. Bohannon, M. Flaster, W. Fan, and R. Rastogi. A cost-based model and effective heuristic for repairing constraints by value modification. In *SIGMOD*, pages 143–154, 2005.
- [3] G. Cong, W. Fan, F. Geerts, X. Jia, and S. Ma. Improving data quality: Consistency and accuracy. In *VLDB*, pages 315–326, 2007.
- [4] Y. Huhtala, P. P. J. Kinen, and H. Toivonen. Efficient discovery of functional and approximate dependencies using partitions. *ICDE*, pages 392–401, 1998.
- [5] N. Prokoshyna, J. Szlichta, F. Chiang, R. Miller, and D. Srivastava. Combining quantitative and logical data cleaning. *PVLDB*, 9(4):300–311, 2015.
- [6] C. Wyss, C. Giannella, and E. L. Robertson. FastFDs: Heuristic-driven, depth-first alg. for mining FDs from relations. In *DaWaK*, pages 101–110, 2001.

Analysis and Visualization of Urban Emission Measurements in Smart Cities

Dirk Ahlers
NTNU – Norwegian University of
Science and Technology
dirk.ahlers@ntnu.no

Frank Alexander Kraemer
NTNU – Norwegian University of
Science and Technology
kraemer@ntnu.no

Anders Eivind Braten
NTNU – Norwegian University of
Science and Technology
anders.e.braten@ntnu.no

Xiufeng Liu
DTU – Technical University of
Denmark
xiuli@dtu.dk

Fredrik Anthonisen
AIA Science AS
Trondheim, Norway
fredrik.anthonisen94@gmail.com

Patrick Driscoll
NTNU – Norwegian University of
Science and Technology
patrick.arthur.driscoll@ntnu.no

John Krogstie
NTNU – Norwegian University of
Science and Technology
john.krogstie@ntnu.no

ABSTRACT

Cities worldwide aim to reduce their greenhouse gas emissions and improve air quality for their citizens. Therefore, there is a need to implement smart city approaches to monitor, model, and understand local emissions to better guide these actions. We present our approach that deploys a number of low-cost sensors through a wireless Internet of Things (IoT) backbone and is thus capable of collecting high-granular data. Based on a flexible architecture, we built an ecosystem of data management and data analytics including processing, integration, analysis, and visualization as well as decision-support systems for cities to better understand their emissions. Our prototype system has so far been tested in two Scandinavian cities. We present this system and demonstrate how to collect, integrate, analyze, and visualize real-time air quality data.

1 INTRODUCTION

Urban emissions contribute over 60% to global greenhouse gas emissions. Cities aim at reducing their emissions through tailored policy and integration to Smart City approaches. Smart City approaches facilitate easier integration of emission sensing into city systems and fulfill city requirements through novel and low-cost approaches [5, 6, 8, 10, 11].

The overall aim of our project¹ is to fulfill the information needs of cities that need specific data for emission reduction actions by providing complementary on-the-ground emission data for improved understanding and decision making [2]. In short, the need based on future challenges faced by cities will be better and more high-granularity measurements to complement existing official measurement stations.

Some Nordic cities have specific challenges in that they have already implemented a range of climate actions, which means that future impact on a certain class of emissions can only be achieved by a more detailed and granular understanding and analysis of emissions, since many broad measures are already in place. The next step then is to get better insight into more

difficult to measure components, also to be able to adapt policy in fast feedback loops and at varying scales. This includes impact assessment of measures ranging from small-scale such as closing down certain streets (and being able to observe spillover and evasion effects in surrounding parts of the city) to large-scale such as changes in public transport or denser urban development.

A high spatial granularity of sensor deployments is obviously not possible with the existing expensive high-quality measurement stations that are often provided nationally. Our approach, in contrast, is to use low-cost sensors to cover a city's spatial footprint with a much higher sensor density. This enables a trade-off of high number and high granularity of low-cost sensors that can compensate for their relatively lower accuracy.

Existing official measurement stations are equipped with high-quality sensors that cost up to \$500,000. Our low-cost approach could provide a very dense coverage of a city with 250 additional sensors for the price of one additional station by using sensor units of around \$2,000 each. For ease of installation, this requires standalone sensor units that do not need cabling for electricity or connectivity. We achieve that by deploying solar-powered sensor nodes with a wireless data link over the LoRaWAN standard for Smart City IoT applications, which also enables us to quickly scale up the sensor deployment. The approach allows to quickly prototype system components on the hardware and software side for the overall goal of linking the measurement data to the information needs of the cities for emission reduction both for baseline and continuous data collection.

After having built and deployed the general IoT sensor network before [2], we focus here on the integration of data sources and the data analysis infrastructure for Smart City applications.

2 APPROACH

Our approach is to build an ecosystem of relevant tools and methods to better understand city emissions and work with data, such as analytics [9, 12], visualizations [7], and decision support systems [5, 6, 11] around local emission measurements and the integration of external data sources. This is an important aspect of Smart Cities [9], and can also be used as a case study to understand and build similar systems. Our system is piloted in the two cities of Trondheim, Norway, and Vejle, Denmark.

In this paper, we describe key aspects of this ecosystem of data analysis and visualization that strongly relates to challenges and

¹Carbon Track & Trace – CTT: <https://www.ntnu.edu/ad/ctt>

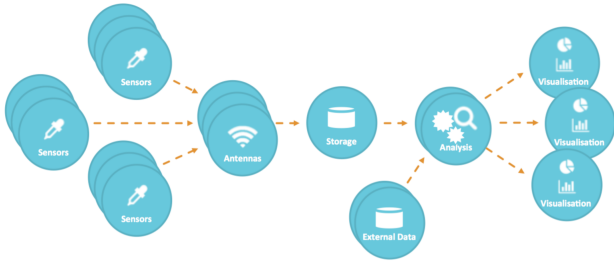


Figure 1: Overall system architecture

requirements of the cities. We further demonstrate the integration and aggregation of data sources for a smart city.

2.1 Architecture

The system architecture and data flow is sketched in Fig. 1, which consists of four components: a city-wide IoT sensor network, cloud-based systems for data collection and storage, integration of external data, and analysis and visualization platforms for stakeholders. The architecture is flexible through an ecosystem approach and accommodates different components for a range of related tasks. Our technology stack follows common concepts for IoT and Smart City systems [10] with project-specific adaptations.

The sensor network is composed of sensor nodes deployed within the city, which measure emissions and air parameters: CO₂, NO₂, PM_x (particulate matter); temperature, pressure, and humidity. The data is transmitted to the IoT backbone, which forwards collected data to the cloud storage, from where it is available for analysis and visualization, using relevant external data sources. The backbone uses LoRaWAN as a radio-based urban sensor networks through a number of gateways covering the pilot regions [2]. Data forwarding and cloud sensor management was built through the event-driven MQTT communication protocol.

Visualizations and analyses are connected to all stages of the data processing. Examples are network monitoring and early data validation close to the sensors, stream processing on measurement data, up to C&C centers, satellite measurement grounding, integration into GML-based 3D city models, and other forms of mapping and integration that we describe in the following.

2.2 Data Integration

Apart from the direct sensor data, there is a range of municipal and national data sets available as well as other external data sources that need to be included in the data analytics and visualization to support analyses and improve data quality. Table 1 gives an overview of these sources and how they can be utilized. They range from direct measurements of air quality that can be used to validate and calibrate the sensor network to other data sources that help to understand emissions in the context of a city, for example through traffic patterns [12] or integration into city tools and systems.

The sensor network has the usual issues of missing data that is dealt with on a technical monitoring level and being handled by standard methods in the analyses, as well as the aggregation of data from multiple sensor units. More interesting are the challenges posed in the data integration. The sources contain highly heterogeneous data, with different timescales, measurement frequencies, spatial distributions and granularities, measurement technologies, and a complex set of related uncertainties and inaccuracies in the data.

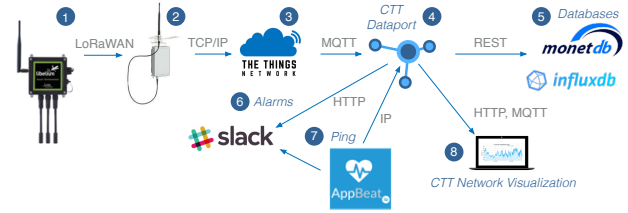


Figure 2: Dataport Protocol Diagram

2.3 Network Metadata Analysis and System Status Monitoring

The network, server components, gateways and sensors are subject to transient and permanent failures, which can ultimately result in missing data. Although the later analysis tasks can detect such losses of data, they do not analyze the cause for the error, or prevent further losses. Instead, failures in the system should be detected as quickly as possible, so that data loss is kept at a minimum. We therefore built a monitoring application (the *dataport*) to monitor the status of all sensors, gateways and the network [3]. It is built with the Akka framework, which facilitates the creation of fault-tolerant applications based on the actor model [4]. Actors are independent, supervised processes that encapsulate data and control logic and communicate via messages. Each device in the real world corresponds to a dedicated actor that acts as its digital twin, which is a virtual model of the sensor or gateway. It keeps track of its state in real-time, monitors all communication and triggers alarms if data is not received as expected. Incoming data contains meta-data that identifies the originating sensor and the gateway from which it was received. In this way, the digital twin for a gateway can detect if a gateway operates as expected.

Faults of a more complex nature, such as decaying sensors, erroneous behavior of sensor nodes, or missing data patterns need specific analysis. For example, a single missing measurement is expected occasionally. Based on the measurement frequency of individual sensors, it takes some cycles to determine a failure with certainty. As sensors nodes can adapt their frequency based on battery levels, a complex model of the sensor node and its status is needed for detection. Actors are organized hierarchically. On higher levels, failures can be grouped so that for example a distinction can be drawn between sensor failures versus a gateway outage that would make a set of sensors invisible.

The dataport also monitors the larger system, such as the The Things Network (TTN) cloud backend and the MQTT connection. If any of the components on the data path from the sensors to the data storage fails, the dataport generates a notification. If the dataport itself fails, it is detected by an external watchdog service, in this case AppBeat. The dataport further drives a visualization of the network itself, shown in Fig. 3, of the structure of digital twins for sensors and gateways, their location, the connections and live data transmission between sensors and gateways. Apart from the practical value of monitoring the network, it is also a useful illustration of the spatial and measurement characteristics.

2.4 Data Analyses and Visualizations

A range of analyses work on the collected data streams as illustrated in Fig. 1 apart from the more operational network analysis. Examples are ongoing data collection and analysis, understanding of patterns, as well as comparison of sensor measurements to air quality measurement stations to ground the network and calibrate the sensors. There are very few official stations; to

Table 1: Examples of external data integration

Type	Example	Description
Official air quality measurements	NILU data (Norwegian Air Quality Institute)	Ground truth for certain pollution types, grounding and calibrating measurements to high-quality reference stations
Remote sensing	NASA OCO-2 satellite CO ₂ measurements	Ground truth top-down measurements for certain emission types, large-scale coverage, low spatial resolution, coupling to large-scale modeling and validation
Traffic data	Traffic density from here.com	Estimate traffic emissions by correlating continuous external traffic density to emission measurements
3D city models	Municipal traffic counts Municipal 3D model of Vejle	Validate traffic estimations, but only available for short periods Integration into existing visualization tools. Use of city geometry in future emission modeling
National statistics	GHG emission estimates from national statistics office	down-scaled national GHG emission data, often with high uncertainties
Other municipal data and tools	GIS, statistics, decision support, etc.	Understanding emissions in the context of the city



Figure 3: Visualization of sensors, gateways, and links

support the grounding and calibration, we have co-located one of our sensor units to the only station in the pilot area. This allows to compare both absolute and relative accuracy and calibrate the local sensor and, through larger-scale correlated trends, the network, but with lower certainty. In connection with the network monitoring, it also allows the identification of outliers and malfunctioning sensors. Main ongoing work is modeling dependencies of NO₂, PM_x, and CO₂, especially from transport emissions, which therefore also looks at linking to traffic patterns [12]. We discuss some analytics around this data in the following.

Battery levels depend on the charging of the autonomous sensor units through their solar panels. Charged occurs during daytime, and is affected by weather conditions. It is important to monitor the battery level to keep the nodes running. Fig. 4 shows the battery level as a function of time (left), and the difference in battery-level from previous sent package versus time of day, and where red indicates whether the nodes could have been charged by sunlight since the previous package (right). This allows to estimate battery depletion.

Dynamics of CO₂ emissions and possible links to traffic in the form of a traffic jam factor (from here.com data) is shown in Fig. 5. According to the plots, we can conclude for this sensor location that traffic is not the only factor that accounts for the dynamics of the CO₂ emission as they exhibit different patterns, and have no apparent correlation. In fact, CO₂ emission dynamic is a more complex issue that may be affected by many factors, including traffic, wind speed, temperature, humidity and other weather conditions, as well as daily and seasonal patterns, which we will further investigate in our future work.

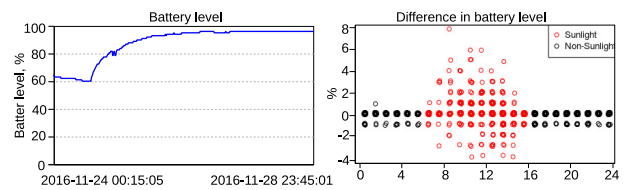


Figure 4: Battery level analysis

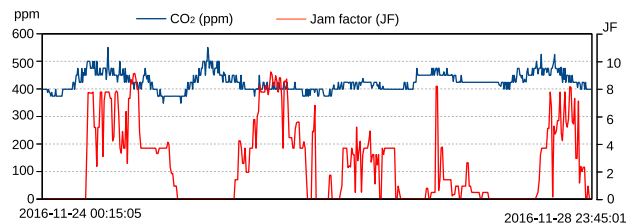


Figure 5: A study of CO₂ dynamics

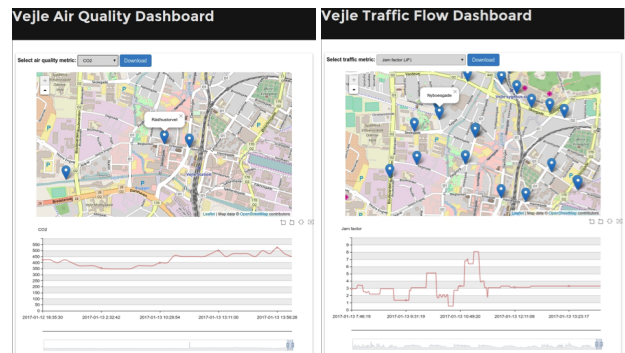


Figure 6: Example of dashboards for air quality and traffic

Visualizations and Dashboards for real-time monitoring. Fig. 6 shows the air quality and traffic flow dashboard, respectively. The dashboard is implemented using Apache Zeppelin as the visualization platform and accesses the data from the OpenTSDB time series database. The mapped sensors show the real-time data and analytic results for each location. Examples are the the air quality and traffic indicators in Fig. 6. This was further integrated into a 3D CityGML model as seen in Fig. 7 and also into a full network and data overview wall display shown in Fig. 8.

3 DEMONSTRATION

In this first full demonstration of the CTT air quality system, we show the architecture and implementations of IoT and analytics

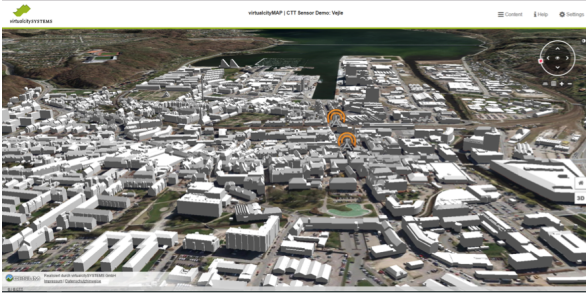


Figure 7: Integration of sensor data into 3D city model

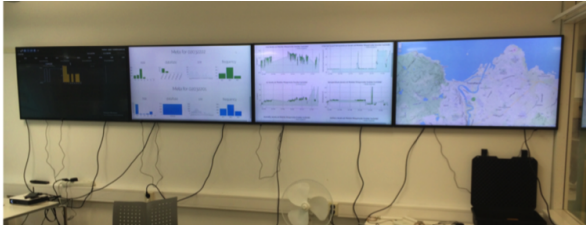


Figure 8: Network monitoring and data visualization dashboards

technologies in air quality monitoring and explore insights. We use two use cases of deploying our systems in Vejle, Denmark and Trondheim, Norway, where two and twelve sensors were deployed respectively to collect air quality data. We demonstrate our system from the perspective of developers, city policymakers, and citizens. For developers, we explore the system in detail, demonstrate the building blocks of the system, and show how to build similar IoT systems; for policymakers, we aim to assist them in decision making for smart cities with the proposed IoT technologies, e.g., urban planning; for citizens, we aim to raise the awareness of environment protection and greenhouse gas reduction for better city life.

We use real-time data collected from the deployed sensors from both cities, as well as traffic data sets streamed from the third-party traffic flow monitoring operator, here.com. The sensor data consist of the CO₂, NO₂, and PM_x, and weather data including humidity and temperature. The sensor data is collected at a five-minute interval. The demo also uses historic data saved in our time-series database, collected since January 2017.

Developers' point of view: We show the architecture and components used by our air quality monitoring system, including sensors, IoT sensor network, cloud storage for sensor data and external traffic data, analytics, and dashboards. We demonstrate how to collect, process and visualize high-frequent sensor data in our system developed on the Zeppelin platform; and how to streamline the whole data flow, including segmentation, chaining, and automation. Finally, we demonstrate how to generate dashboards and integrate analysis algorithms in the web interface. Attendees can vary system and analysis properties, and observe the reflection on the dashboard; and change the dependency of the data flow to evaluate the flexibility of the data stream analysis.

City officials' point of view: We show an interactive dashboard to analyze CO₂ dynamics using real-time and historic measurement data, and demonstrate the pattern and its correlation to the traffic flow (see Fig. 5–6). In addition, we demonstrate the 3D CityGML model integrating different measuring points of air quality (see Fig. 7). In this demo scenario, we can inject synthetic data showing different pollution levels. We interact with attendees by discussing urban planning issues such as construction

sites of roads, buildings or factories, and see how different pollution levels will affect their decision makings. Also, we consult with attendees about choosing the sites of air quality monitoring, e.g., according to the road network and building density.

Citizens' point of view: We demonstrate air quality and traffic flow on the dashboard using the real-time data. Similarly, we use synthetic data with different pollution levels, and discuss the influence on routing planning, and citizens' approaches for emission reduction. Attendees can browse historic data in the system to investigate anomalous emission levels.

4 CONCLUSION

We have described the possibilities for urban emission monitoring and our approach and the prototype system we have developed together with the approaches to data flows and analysis. The flexible and scalable solution allows to quickly prototype different analysis approaches on top of the sensor streams to link measured data to cities' information needs for emission reduction.

In future work, we plan to improve the measurement network and the real-time and aggregate dashboards. Further, with more data collected, we will be able to tune models for emission distribution and dispersion to overcome some of the issues and provide improved analysis with better models. Integration into decision support systems is a far goal. Urban emission monitoring needs a range of heterogeneous data and we are continuing to build useful urban systems around it.

ACKNOWLEDGEMENTS

This paper is part of the Carbon Track & Trace (CTT) project. Funding for this project was in part provided by the LoCaL² Climate-KIC flagship programme. We thank all our project partners and specifically VCS³ for the 3D city model demo. Parts of this article are based on project deliverables [1].

REFERENCES

- [1] Dirk Ahlers and Patrick Driscoll. 2016. *Technical Architecture. CTT2.0 Deliverable*. Technical Report D2.4. NTNU.
- [2] Dirk Ahlers, Patrick Driscoll, Frank Alexander Kraemer, Fredrik Anthonisen, and John Krogstie. 2016. A Measurement-Driven Approach to Understand Urban Greenhouse Gas Emissions in Nordic Cities. In *NIK2016 – Norwegian Informatics Conference*.
- [3] Hans Henrik Grønsløth. 2016. *Urban Greenhouse Gas Level Monitoring Using LoRaWAN*. Master's thesis. NTNU.
- [4] Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *IJCAI'73*. 235–245.
- [5] François Ingelrest, Guillermo Barrenetxea, Gunnar Schaefer, Martin Vetterli, Olivier Couch, and Marc Parlange. 2010. SensorScope: Application-specific Sensor Network for Environmental Monitoring. *ACM Trans. Sen. Netw.* 6, 2, Article 17 (2010).
- [6] Prashant Kumar, Lidia Morawska, Claudio Martani, George Biskos, Marina Neophytou, Silvana Di Sabatino, Margaret Bell, Leslie Norford, and Rex Britter. 2015. The rise of low-cost sensing for managing air pollution in cities. *Environment International* 75 (2015), 199–205.
- [7] Jonathan Liono, Flora Dilys Salim, and Irwan Fario Subastian. 2015. Visualization Oriented Spatiotemporal Urban Data Management and Retrieval. In *UCUI'15 (CIKM'15)*. ACM.
- [8] Kirk Martinez, Jane K. Hart, and Royan Ong. 2004. Environmental sensor networks. *Computer* 37, 8 (2004), 50–56.
- [9] Johannes M. Schleicher, Michael Vögler, Christian Inzinger, and Schahram Dustdar. 2015. Towards the Internet of Cities: A Research Roadmap for Next-Generation Smart Cities. In *UCUI'15 (CIKM'15)*. ACM.
- [10] A. Zanello, N. Bui, A. Castellani, L. Vangelista, and M. Zorzi. 2014. Internet of Things for Smart Cities. *IEEE Internet of Things Journal* 1, 1 (Feb 2014), 22–32.
- [11] Yu Zheng, Furu Liu, and Hsun-Ping Hsieh. 2013. U-Air: When Urban Air Quality Inference Meets Big Data. In *KDD '13*. ACM.
- [12] Zimu Zheng, Dan Wang, Jian Pei, Yi Yuan, Cheng Fan, and Fu Xiao. 2016. Urban Traffic Prediction Through the Second Use of Inexpensive Big Data from Buildings. In *CIKM '16*. ACM.

²<http://local.climate-kic.org/>

³<https://www.virtualcitysystems.de/>

Pharos: Privacy Hazards of Replicating ORAM Stores

Victor Zakhary, Cetin Sahin, Amr El Abbadi, Huijia (Rachel) Lin, Stefano Tessaro
 University of California, Santa Barbara, CA 93106
 [victorzakhary,cetin,amr,rachel.lin,tessaro]@cs.ucsb.edu

ABSTRACT

Although outsourcing data to cloud storage has become popular, the increasing concerns about data security and privacy in the cloud blocks broader cloud adoption. Recent efforts have developed oblivious storage systems to hide both the data content and the data access patterns from an untrusted cloud provider. These systems have shown great progress in improving the efficiency of oblivious accesses. However, these systems mainly focus on privacy without considering fault-tolerance of different system components. This makes prior proposals impractical for cloud applications that require 24/7 availability. In this demonstration, we propose **Pharos**, the **Privacy Hazards of Replicating ORAM Stores**. We aim to highlight the data access pattern privacy hazards of naively applying common database replication and operation execution techniques such as locking and asymmetric quorums.

1 INTRODUCTION

Outsourcing data to cloud storage has become increasingly popular due to its promise of better scalability and availability. However, existing confidentiality concerns [6] make potential users often skeptical about joining the cloud. While necessary, encryption alone is not sufficient to solve all privacy challenges posed by outsourcing private data. Although encryption hides the data content, the cloud provider can monitor the *access patterns* of each data block. This includes the recency and frequency of access in addition to the type of access such as reads and writes of each data block. Revealing data access patterns presents a real threat to the privacy of the outsourced data. Data access patterns can leak sensitive information using prior knowledge. For example, Islam et al. [11] show a concrete inference attack against an encrypted e-mail repository exploiting only access patterns.

Oblivious RAM (ORAM) – a cryptographic primitive originally proposed by Goldreich and Ostrovsky [9] as a solution for software protection – is the standard approach to hide access patterns. ORAM aims to hide the *target block of each access operation* and the *access operation type*. To achieve these *two* goals, each logical access is translated to a sequence of random looking accesses. In addition, ORAM shuffles and re-encrypts the data content in each data access, making access patterns of any two equally long sequences of read/write operations completely indistinguishable.

Hiding access patterns was initially considered in the context of memory access [9]. While classical ORAM schemes with small client memory apply directly to the memory access setting, in cloud applications a client has more storage space and is capable of storing more data locally and more importantly can outsource the storage of a large dataset to the cloud. The benefits, as well as, the fast adoption of the cloud encouraged the research community in the past several years to develop new secure data services. Many ORAM schemes have been constructed for secure cloud

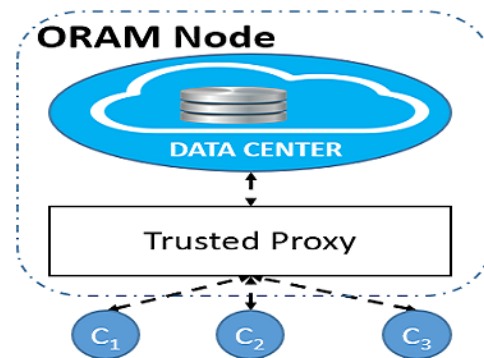


Figure 1: An ORAM node and clients.

storage systems [3, 4, 12–15, 17]. The initial ORAM proposals focused on developing ORAM constructions for a single client setting. These systems do not fit the requirements of cloud deployments, since accesses to the storage are performed sequentially (e.g. [9, 16]). To overcome the limitations of single client ORAM constructions, recent proposals develop new constructions that adopt real-world storage requirements by enabling multi-client concurrent and asynchronous accesses while preserving access privacy [3, 13, 14]. Figure 1 shows the typical architecture of an ORAM node in a cloud setting. The data is outsourced to the cloud and clients communicate with a trusted proxy to serve their requests. The trust proxy is responsible for data retrieval from the cloud storage in an oblivious manner.

Although these systems have shown great improvements in terms of efficiency and throughput, to the best of our knowledge, none of the earlier oblivious cloud storage systems were designed to tolerate the failure of different system components. Designing systems that run on commodity machines should consider failures of different system components as the norm such as machine crashes and network partitioning [7]. By ignoring fault tolerance as an important design principle, existing oblivious cloud storage solutions do not fit the requirements of real-world applications.

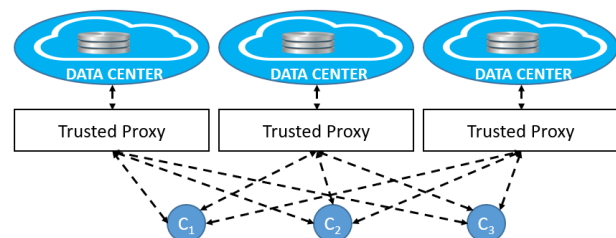


Figure 2: Replicated Cloud ORAM Node.

This demonstration, Pharos, is an initial attempt to illustrate the hazards of ignoring privacy when blindly applying standard database techniques to ensure fault-tolerance. The standard database approach for handling concurrency is locking. The standard

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

approach to ensure fault-tolerance is replication, and typically quorums are used, where quorums of sets of replicas are chosen so that any two quorums have a non-empty intersection. In a database system, read operations are expected to be much more frequent than write operations, hence in a typical database system read operations acquire shared read locks while write operations acquire exclusive write locks. Furthermore, in a replicated fault-tolerant setting, asymmetric quorums are typically used, i.e., read operations use read quorums and write operations use write quorums, where read quorums do not need to intersect, but should intersect with write quorums. Since read operations dominate database workloads, read quorums are typically much smaller than the write ones. In fact, if write availability is not critical, a read quorum of one replica and a write quorum of all replicas is often recommended (or to increase write fault-tolerance, a read quorum of two replicas and a write of all replicas minus one).

Pharos supports oblivious access to replicated cloud ORAM nodes. Figure 2 shows an example of a replicated cloud ORAM node. The trusted proxy and the cloud storage are replicated to tolerate the failure of different system components (the cloud or the trusted proxy). Clients independently communicate with different trusted proxies to serve their read and write operations. During the demo, the attendees will be challenged to support a concurrency control and replication scheme. The traditional approaches of locking and asymmetric quorums will be proposed and contrasted with lock-free and symmetric quorums. Either the attendee or the system will propose an attack to illustrate the possible data access privacy violations in the chosen solutions. Pharos is a demonstration that will help bridge the gap between the database community and the security and privacy community by highlighting the privacy hazards in standard database solutions. Pharos presents attendees different attacks, thus giving them a real sense of the potential hazards of using a replication technique or an operation execution technique on the privacy of data access patterns.

2 DATA MODEL

We give a brief overview of oblivious storage systems that depend on a trusted proxy [3, 13, 14]. As shown in Figure 1, an **ORAM node** consists of a *storage* service that is outsourced to the cloud and a *trusted proxy* that is deployed in between to mediate client server communication as well as to execute the oblivious algorithm. Data is stored in a key-value store, which is encrypted and outsourced to the storage and the meta-data used to locate objects in the storage is maintained in the trusted proxy. Clients submit single operations: object lookups, $get(k)$, and object updates, $put(k, v)$, to the trusted proxy, where k is a key, and v is a value. The trusted proxy translates these requests into oblivious retrievals (OR) and oblivious evictions (OE). An **oblivious retrieval** translates a client get or put of an object O into fetching multiple objects where accessing O is obfuscated among the fetched objects. The trusted proxy has to write-back the retrieved objects by performing an oblivious eviction. An **oblivious eviction** hides the type of client access, (get or put), and the access frequency of different objects by shuffling and re-encrypting all the fetched objects at the trusted proxy before writing them back to the storage. Also, the trusted proxy has to update its meta-data to be able to locate these objects in the storage for later accesses.

3 THREAT MODEL

We consider the threat model for asynchronous ORAM that is introduced in [13]. The threat model assumes an honest-but-curious adversary which can see the raw storage and network communication of the server. It controls the asynchronous links where arbitrary delays can be added to different communication messages. Additionally, an adversary can adaptively schedule access, read and write operations and learn the timing of the responses. This security definition is called *aaob-security*. It formalizes the obliviousness in asynchronous and concurrent multiple access deployment scenarios and ensures that two timing consistent executions should be indistinguishable in the described threat model.

4 PRIVACY HAZARDS OF TYPICAL DATABASE TECHNIQUES

Replication has always been used to provide fault tolerance to database systems. Many of the database replication techniques were developed to focus on performance and data consistency assuming that all the data replicas are trusted. Designing systems where privacy is considered a first class requirement narrows the design choices. Many common design alternatives that enhance the performance might lead to a data access pattern privacy violation. We first illustrate the privacy hazards of using *asymmetric read and write quorums*. Then we demonstrate the problems of using *lock-based concurrency control* solutions.

4.1 Asymmetric Quorums Hazard

Database replication is commonly used to tolerate server failures. Once data is replicated, consistency becomes an important challenge. We assume linearizability [10] as a correctness condition for object accesses. Reading an object should always return the most recent committed update to this object. Although replicas can have different versions of the same object, client reads should always return the latest value of an object. This behavior is defined as operation consistency [2]. Operation consistency requires that clients receive the correct expected results regardless of the state consistency of replicas.

Different replication strategies introduce a trade off between fault tolerance, *the number of replica failures f that the system can tolerate before it stops completely*, and performance, *the number of replicas that should be accessed per read and update operations* for a given consistency requirement. We present the trade offs of different replication strategies.

Linearizability and operation consistency on the object level are achieved using quorums and version numbers [8]. A read quorum (q_r) is the minimum number of replicas that need to be accessed to retrieve the latest value of an object. A write quorum (q_w) is the minimum number of replicas that need to be updated to guarantee consistent reads. To achieve operation consistency, any read quorum should intersect with all write quorums $q_r \cap q_w \neq \emptyset$. This intersection guarantees that a read will always access the latest value of an object. To achieve total order on updates, a centralized sequencer can be used to assign total order version numbers for updates. In this case, write quorums do not have to intersect (e.g. write one read all). However, a total order can be achieved in a distributed way using quorums. In this case, any two write quorum q_{w_1} and q_{w_2} should intersect in at least one replica $q_{w_1} \cap q_{w_2} \neq \emptyset$. This intersection guarantees that objects will be updated in the same order in all the quorum replicas.

Table 1: Summary of different replication strategies, their requirements, and their guarantees.

Replication Strategy	$ q_r $	$ q_w $	f	Hazardous?
Master/Slave (read optimized)	1	N	0	Yes
Master/Slave (write optimized)	N	1	0	Yes
Majority quorums	N/2	N/2	N/2 - 1	No
Grid quorums [5]	\sqrt{N}	$2\sqrt{N}$	$\sqrt{N} - 1$	Yes
Tree quorums [1]	$\log(N)$	$\log(N)$	$\log(N) - 1$	No

Table 1 summarizes the commonly used quorum sizes and their degree of fault tolerance in the worst case. Majority quorums tolerate the failure of any number of replicas less than a majority. It executes read and write operations from majority quorums. Master/slave (read optimized) requires write quorums of size N. If updates are sent to all the replicas, reading from any single replica returns the most up to date version of an object. Similarly, master/slave(write optimized) requires read quorums of size N and write quorums of size 1. Reading all the replicas of an object guarantees freshness given that updates are applied to at least one replica. In a master/slave model, the failure of one replica halts any update(read optimized) or read(write optimized) and hence stops the whole system. Many proposals are based on the notion of logical structures that are imposed on the replicas and used to reduce the cost of quorums, while maintaining the intersection property. As a result, usually, one operation, eg. read operations, requires smaller quorums and hence is cheaper, but then write operations are more expensive. Cheung et al. [5] proposed the grid protocol to maintain replicated data. Replicas are ordered in a grid and a read quorum is any row **or** any column in the grid and a write quorum is any row **together with** any column in the grid. The failure of a full row or a full column halts the system and prevents any updates. Agrawal and El Abbadi [1] present tree quorums where read quorums and write quorums are paths in the tree from the root to a leaf. The failure of a full path stops the system.

The last column of Table 1 indicates if a replication strategy could introduce any privacy hazards on the data access patterns. As shown, all replication strategies that use *symmetric* read and write quorums are safe while others that use *asymmetric* read and write quorums are hazardous.

Recall that in the threat model of Section 3, an adversary monitors all the communication links between the clients and the trusted proxies. When asymmetric read and write quorums are used, the adversary can easily distinguish between read and write accesses by observing the number of messages sent per access. This allows the adversary to identify the type of access for every legitimate client request. Although asymmetric quorums can benefit the system performance, they can easily be used to leak the access type which represents a privacy hazard on the data access patterns.

4.2 Locking Hazards

All trusted-proxy-based ORAM systems assume a single trusted proxy that serves all the client requests. This single trusted proxy is responsible for hiding the access patterns of all client requests and achieving linearizability of access for every data object. Upon

replicating the ORAM nodes and their trusted proxies, the problem of consistently updating the data replicas occurs. In particular, concurrent updates on the same object need to be serialized across the different data replicas. A widely used approach to tackle this problem is *locking*. When a client updates a data object, she first acquires an exclusive lock on this object from a write quorum. Upon receiving the lock response from a write quorum, the client updates the data object and releases the locks. This prevents any inconsistencies that may occur from concurrent updates on the same object. Any concurrent access on the same object that is issued by another client has to wait until the locks are released. However, this concurrent access would not block if it updates a different object.

Recall that in the threat model of Section 3, an adversary can adaptively schedule read and write accesses while trying to understand the objects that are being accessed by other clients. When locks are used, the adversary can concurrently schedule write accesses on all the objects immediately after a legitimate client's access. The adversary should receive the requested locks on all the objects except the one that is being accessed by the legitimate client. By observing the locking responses, the adversary can identify which object is being accessed by the legitimate client request. This attack illustrates the hazard of using locks on the privacy of data access patterns.

5 PHAROS

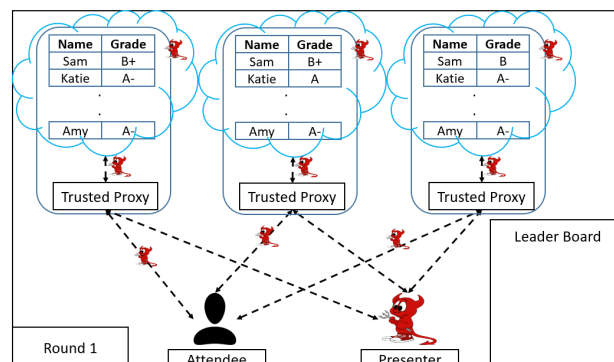


Figure 3: An adversary tries to understand the request types and the data records accessed by a legitimate client.

We propose Pharos to motivate a deeper understanding and appreciation of the privacy hazards that result from the naive application of locks and asymmetric quorums to replicate ORAM stores. Both the attacks on locks and asymmetric quorums result from observing only the communication patterns between the client and the trusted proxy. These attacks are independent of the ORAM implementation. Therefore, we assume a simplified system implementation to illustrate the attacks without distracting the attendee with the ORAM implementation details.

The setup: we assume a simple dataset of 10 student records stored in the ORAM node. Each record has a student name and a grade. The dataset is replicated into three trusted-proxy-based ORAM nodes for fault tolerance. These ORAM nodes are hosted on three cloud machines. The attendee gets to know that locks are used to ensure the consistency of the updates. Also, asymmetric read one write all quorums are used to optimize the latency of read operations. Recall that in the data model of Section 2,

a client can submit $get(student_name)$ to read the record of a student and $put(student_name, grade)$ to update the grade of a student. There are two types of players in the system: legitimate clients and adversaries. A legitimate client submits requests to read or update a data record. An adversary monitors the network, adds delays to messages, and schedules read and write requests in an attempt to reveal the data record or the type of request submitted by a legitimate client. To simulate adversarial network monitoring, each ORAM node sends the adversary the number of messages it receives and an identification of the source of these messages. The setup and the adversary capabilities are shown in Figure 3.

This demonstration proceeds in multiple rounds where the attendee gets to play one of the two roles and the presenter plays the other role. *In the first round*, the attendee is asked to update some student record and the presenter, on another machine, tries to reveal the data record. *In the second round*, the attendee and the presenter flip roles. The presenter submits a sequence of read and update requests and the attendee tries to reveal the type of request submitted by the presenter. The attendees who are able to explain how the two attacks work will have their names written on the *leader board*.

6 CONCLUSION

Private access of data stores is becoming very relevant in the context of cloud computing. Prior work in the security community focused on the privacy of access. We strongly believe that for the success of the cloud, **both** privacy and fault-tolerance dimensions need to be addressed. The goal of Pharos is twofold. First we propose some simple but effective attacks that reveal significant security leaks when the standard approaches for replication are incorporated in a straightforward manner with the privacy preserving solutions. Second, Pharos is a pedagogical game that educates the attendee on the privacy hazards of replication. Our aim is to make the database practitioner attendee aware of these challenges and hence be motivated to better understand the privacy concerns when designing fault-tolerant database systems.

7 ACKNOWLEDGEMENT

This work is partially supported by NSF grants CNS-1528178 and CNS-1649469.

REFERENCES

- [1] Divyakant Agrawal and Amr El Abbadi. 1991. An efficient and fault-tolerant solution for distributed mutual exclusion. *ACM Transactions on Computer Systems (TOCS)* 9, 1 (1991), 1–20.
- [2] Marcos K Aguilera and Douglas B Terry. 2016. The Many Faces of Consistency. *IEEE Data Eng. Bull.* 39, 1 (2016), 3–13.
- [3] Vincent Bindschaedler, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, and Yan Huang. 2015. Practicing Oblivious Access on Cloud Storage: The Gap, the Fallacy, and the New Way Forward (CCS '15). ACM, 837–849. DOI: <http://dx.doi.org/10.1145/2810103.2813649>
- [4] Dan Boneh, David Mazieres, and Raluca Ada Popa. 2011. *Remote Oblivious Storage: Making Oblivious RAM Practical*. Technical Report. MIT. MIT Tech-report: MIT-CSAIL-TR-2011-018.
- [5] Shun Yan Cheung, Mostafa H Ammar, and Mustaque Ahamad. 1992. The grid protocol: A high performance scheme for maintaining replicated data. *IEEE Transactions on Knowledge and Data Engineering* 4, 6 (1992), 582–592.
- [6] Richard Chow, Philippe Golle, Markus Jakobsson, Elaine Shi, Jessica Staddon, Ryusuke Masuoka, and Jesus Molina. 2009. Controlling Data in the Cloud: Outsourcing Computation Without Outsourcing Control. In *Proceedings of the 2009 ACM Workshop on Cloud Computing Security (CCSW '09)*. ACM, New York, NY, USA, 85–90. DOI: <http://dx.doi.org/10.1145/1655008.1655020>
- [7] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: amazon's highly available key-value store. *ACM SIGOPS operating systems review* 41, 6 (2007), 205–220.
- [8] David K Gifford. 1979. Weighted voting for replicated data. In *Proceedings of the seventh ACM symposium on Operating systems principles*. ACM, 150–162.
- [9] Oded Goldreich and Rafail Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *J. ACM* 43, 3 (May 1996), 431–473. DOI: <http://dx.doi.org/10.1145/233551.233553>
- [10] Maurice P Herlihy and Jeannette M Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 3 (1990), 463–492.
- [11] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2012. Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*.
- [12] Jacob R. Lorch, Bryan Parno, James Mickens, Mariana Raykova, and Joshua Schiffman. 2013. Shroud: Ensuring Private Access to Large-Scale Data in the Data Center. In *USENIX FAST '13. USENIX*, 199–213. <https://www.usenix.org/conference/fast13/technical-sessions/presentation/lorch>
- [13] C. Sahin, V. Zakhary, A. El Abbadi, H. Lin, and S. Tessaro. 2016. TaoStore: Overcoming Asynchronicity in Oblivious Data Storage. In *2016 IEEE Symposium on Security and Privacy (SP)*. 198–217. DOI: <http://dx.doi.org/10.1109/SP.2016.20>
- [14] Emil Stefanov and Elaine Shi. 2013. ObliviStore: High Performance Oblivious Cloud Storage. In *IEEE SP '13*. 253–267. DOI: <http://dx.doi.org/10.1109/SP.2013.25>
- [15] Emil Stefanov, Elaine Shi, and Dawn Xiaodong Song. 2012. Towards Practical Oblivious RAM. In *NDSS '12*. <http://www.internetsociety.org/towards-practical-oblivious-ram>
- [16] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: An Extremely Simple Oblivious RAM Protocol (CCS '13). ACM, 299–310. DOI: <http://dx.doi.org/10.1145/2508859.2516660>
- [17] Peter Williams, Radu Sion, and Alin Tomescu. 2012. PrivateFS: A Parallel Oblivious File System (CCS '12). ACM, 977–988. DOI: <http://dx.doi.org/10.1145/2382196.2382299>