Optimization of Complex SPARQL Analytical Queries

Padmashree Ravindra[*]
Microsoft Corporation, Redmond, USA
paravin@microsoft.com

HyeongSik Kim, Kemafor Anyanwu
North Carolina State University, Raleigh, USA
{hkim22, kogan}@ncsu.edu

ABSTRACT

Analytical queries are crucial for many emerging Semantic Web applications such as clinical-trial recruiting in Life Sciences that incorporate patient and drug profile data. Such queries compare aggregates over multiple groupings of data which pose challenges in expression and optimization of complex grouping-aggregation constraints. While these challenges have been addressed in relational models, the semi-structured nature of RDF introduces additional challenges that need further investigation. Each grouping required in an RDF analytical query maps to a graph pattern subquery with related groups leading to overlapping graph patterns within the same query. The resulting algebraic expressions for such queries contain large numbers of joins, groupings and aggregations, posing significant challenges for present-day optimizers.

In this paper, we propose an approach for supporting efficient and scalable RDF analytics that follows the well known technique of simplifying algebraic expressions of RDF analytical queries in a way that enables better optimization. Specifically, the approach is based on a refactoring of analytical queries expressed in the relational-like SPARQL algebra based on a new set of logical operators. This refactoring achieves shared execution of common subexpressions that enables parallel evaluation of groupings as well aggregations, leading to reduced I/O and processing costs, particularly beneficial for scale-out processing on distributed Cloud systems. Experiments on real-world and synthetic benchmarks confirm that such a rewriting can achieve up to 10X speedup over relational-style SPARQL query plans executed on popular Cloud systems.

1. INTRODUCTION

Growing amount of linked open data is enabling interesting applications that combine data from different domains for analysis. For example, the ReDD-Observatory [38] discusses a study reporting the total number of deaths and the number of clinical trials for Tuberculosis and HIV/AIDS in all countries, to analyze the disparity between biomedical research and the disease burden in developing countries. This study involved information about clinical trials and effectiveness of treatment options from *ClinicalTrials.gov*, statistics about mortality for different countries from the *Global Health Observatory* (GHO), published by the World Health Organization and biomedical research (MEDLINE publications and other life science journals) available in the *PubMed*. The results need to be grouped based on both country and disease, followed by aggregations on the number of clinical trials and deaths due to the concerned disease in each country, using the grouping-aggregation constructs in SPARQL 1.1 [22]. Another Semantic Web application, AlzPharm [26], queries several semantically-linked neuroscience datasets to find information relevant to neurodegenerative diseases, e.g., identify the different groups of drugs used for Alzheimer's Disease when grouped by their molecular targets and clinical usage.

Non-trivial analytical queries require multiple aggregations over different groupings of data, some of which may be related, resulting

[*]Majority of the work was done when the first author was a student at the Department of Computer Science, North Carolina State University

```
SELECT ?country ?feature  ((?sumF ∗ (?cntT – ?cntF)) /
                            (?cntF ∗ (?sumT – ?sumF)) As ?priceRatio)
{{
   SELECT  ?country  (count(?price) As ?cntT)  (sum(?price) As ?sumT)
   {
     ?product  rdf:type        PT18.
     ?offer    bsbm:product    ?product ;
               bsbm:price      ?price ;          GP1
               bsbm:vendor     ?vend .
     ?vend     bsbm:country    ?country .
   }
   GROUP BY ?country
}
{
   SELECT  ?country  ?feature
           (count(?price2) As ?cntF)  (sum(?price2) As ?sumF)
   {
     ?product2  rdf:type              PT18 ;
                bsbm:productFeature   ?feature .
     ?offer2    bsbm:product          ?product2 ;     GP2
                bsbm:price            ?price2 ;
                bsbm:vendor           ?vend2 .
     ?vend2     bsbm:country          ?country .
   }
   GROUP BY ?country ?feature
}}
```

Figure 1: (*AQ1*): An example SPARQL analytical query, *For each country, retrieve product features with the highest ratio between price with that feature and price without that feature*

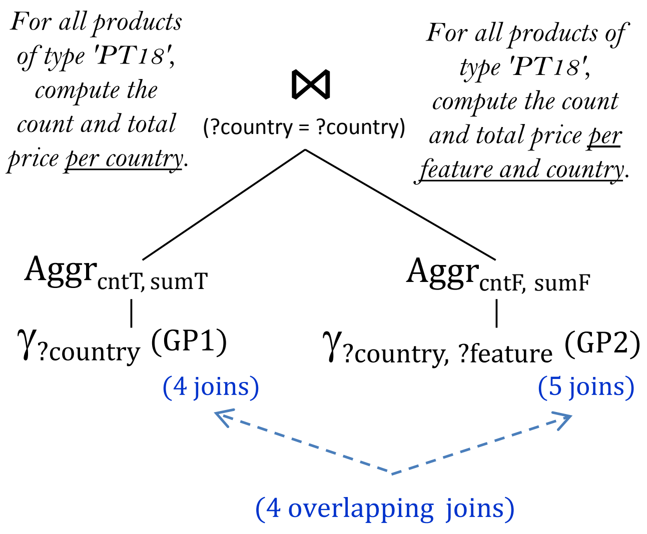

For all products of type 'PT18', compute the count and total price per country.
⋈
(?country = ?country)
For all products of type 'PT18', compute the count and total price per feature and country.
Aggr cntT, sumT
Aggr cntF, sumF
γ?country (GP1)
γ?country, ?feature (GP2)
(4 joins)
(5 joins)
(4 overlapping joins)

Figure 2: A relational-algebra based query plan for AQ1

in redundant scans and joins over large relations. Consider an example SPARQL analytical query *AQ1* shown in Figure 1, adopted from the Berlin SPARQL BI benchmark [1]. The query involves two descriptions `GP1` and `GP2` for products of type '`PT18`' with grouping constraints on `country` and (`country`, `feature`) combinations, respectively. Each grouping constraint is defined over a graph pattern (a combination of one or more triple patterns[1] that specifies constraints to retrieve relevant subgraphs). Queries with multiple groupings involve multiple graph patterns. Further, if the groupings are related then there is a significant amount of overlap in the graph pattern subqueries. Figure 2 shows a summarized query plan with two major subqueries using the traditional evaluation technique: a subquery for `GP1` with four joins that matches subgraphs about offers for products of type `PT18`, their price and vendor information, followed by a grouping on vendor's *country*. The second subquery contains a similar graph pattern `GP2` with five joins (an extra join due to the addition of product feature) followed by a grouping on `country-feature`. Answers from the two subqueries are then joined to compute the final price ratio, resulting in a total of *10 joins and 2 grouping operations*.

In contrast, in the relational model, such OLAP queries are evaluated over suitably organized (star or snowflake) schemas consisting of n-ary relations. Different optimization strategies ranging from specialized query constructs [20, 9, 10], efficient indexing [31, 37], materialized views [21, 12], and efficient evaluation in distributed data warehouses [7, 6] have been proposed. In the absence of such schema organizations for RDF, a naive approach is to decompose the evaluation into two distinct phases: a graph pattern evaluation phase that constructs a suitable set of n-ary relations, followed by relational-style optimizations. However, such an approach prevents the possibility of optimizations across the two phases, e.g., early projections, partial aggregations, etc. Therefore, a holistic optimization strategy is likely to be more advantageous.

A promising direction is based on the observation made in [10] that relational expressions tightly couple grouping and aggregation specifications, often resulting in complex algebraic expressions that confound query optimizers. The approach in this paper has a similar spirit, i.e., grouping-aggregation specifications in RDF analytical queries are decoupled to optimize subqueries. Further, with a focus to support large scale RDF analytics, the paper overviews how such a query reformulation can be evaluated on Cloud platforms such as MapReduce [17]. The challenges with evaluating complex queries with many join operations have been addressed in several papers [4, 23, 33], and can be summarized as long, expensive execution workflows with multiple I/O and network data transfer phases. Many techniques have been proposed to mitigate these costs by sharing scans and computations [30, 28, 33] during MapReduce-based processing. In this paper, we present a holistic optimization that integrates the work on algebraic optimization of graph pattern queries with algebraic optimization of OLAP queries. Specifically, we make the following contributions:

- An algebraic rewriting of overlapping graph patterns (in a SPARQL analytical query) using a *composite graph pattern* based on common substructures. A decoupled reformulation of the grouping-aggregation definitions in a SPARQL analytical query expressed using a composite graph pattern.
- A set of logical and physical operators for efficient evaluation of a composite graph pattern, as well as parallel evaluation of independent aggregations on a composite graph pattern. The suite of operators and optimizations are integrated into *RAPIDAnalytics*, an extension of Apache Pig.
- A comprehensive evaluation of RAPIDAnalytics using basic and multi-aggregation SPARQL analytical queries on real-world as well as synthetic benchmark datasets.

The rest of the paper is organized as follows: Section 2 provides a background on complex OLAP queries and specific challenges in processing such queries over the RDF data model. Section 3 introduces an algebraic rewriting of SPARQL analytical queries based on a non-relational data model and algebra, followed by formal definitions of newly introduced logical operators. Section 4 describes the physical operators and optimizations to execute such a query plan on MapReduce-based platforms. Section 5 presents the comparative evaluation results between RAPIDAnalytics and other popular approaches, and Section 6 presents concluding remarks.

2. BACKGROUND AND CHALLENGES

2.1 Optimization of Complex OLAP Queries

There has been a body of work to enable better expression and evaluation [20, 19, 10, 6, 11] of complex OLAP queries including introduction of constructs such as the `CUBE BY` [20], grouping sets [9], etc., that allow the user to have a finer control over the grouping and aggregation specifications. An earlier work on *MD-Join* [10] showed that decoupling of the grouping definition and aggregation computations not only allows more succinct expression of complex OLAP queries, but can also eliminate redundant scans and joins over large fact tables.

Parallel / Distributed Evaluation of Relational OLAP Queries. An earlier work on parallel evaluation of aggregates proposed adaptive algorithms [35] to handle a range of grouping selectivities (ratio of result size to input size) across queries. Subsequent research [6] on distributed evaluation of OLAP queries identified optimizations that exploit knowledge about data distributions to reduce the amount of data transfer between the local sites and the centralized coordinator. In the context of MapReduce, an overlapping redistribution scheme [14] was proposed to enable parallel evaluation of correlated aggregations with sliding windows. MR-Cube [29] distributes the cube computation of partially algebraic measures on MapReduce. It also introduced a value-partitioning scheme to deal with

[1]RDF data is modeled as a set of triples = (*Subject*, *Property*, *Object*). A triple pattern is a triple with at least one variable denoted by a leading '?'

reducer-unfriendly (cube) groups that tend to increase the load on a reducer. The work on MR-Cube was integrated into Apache Pig[2] and Apache Hive[3] (`GROUPING SETS`, `CUBE` and `ROLLUP` clauses). Such operations assume the existence of a fact relation on which `CUBE` and other operations can be applied, which does not hold true in the case of the RDF data model where triples are commonly represented as binary or ternary relations.

Expression and Evaluation of RDF Analytical Queries. The RDF Data Cube vocabulary (QB) [16] was provided as a recommendation to enable publication of statistical data in RDF adhering to Linked data principles. The work on Open Cubes vocabulary [18] enables representation of multidimensional data using RDF Schema (RDFS). Other extensions [24] propose a multi dimensional model based on QB to support OLAP queries, mapping them to SPARQL. Recent work on RDF analytics [15] proposed a way to define an analytical schema on RDF graphs and formalize analytical queries over such an analytical schema, by separating the grouping-aggregation definitions, similar to the relational *MD-Join* [10] operator. An earlier work [36] extended Pig's query primitives to support MapReduce based execution of the *MD-Join* operator.

Discussion. The *MD-Join* approach to eliminate redundant scans and joins involving large fact relations, translates to reduction in I/O and network transfer costs in MapReduce-based processing of complex analytical queries. However, specifics of RDF analytics make it challenging to adopt such an approach. Unlike traditional OLAP systems where the fact and dimension tables are available and suitably organized into star or snowflake schema, the fine-grained data model in RDF necessitates several join operations to reassemble the relevant fact and dimension information, e.g., fact relation described by `GP1` requires four join operations. A relational-style query plan that computes the detail relations described by `GP1` and `GP2`, compiles into a lengthy MapReduce execution workflow with 9 map-reduce cycles (one per star-join). Such a sequential execution limits opportunities to share input scans in general. Furthermore, RDF analytical queries often involve (slightly) different join expressions for detail relations (refer to `GP1` and `GP2`). Thus, in order to fully exploit the benefit of a decoupled reformulation using the *MD-Join* approach, we require additional optimizations that enable shared execution of the graph patterns in an RDF analytical query.

2.2 Shared Execution of Graph Pattern Queries

A commonly occurring pattern in OLAP queries involves comparing subtotals across multiple dimensions, which results in subqueries that compute groupings over an overlapping subset of dimensions, e.g., `GP2` computes groupings on `country-feature`, while `GP1` is a roll-up on ALL features. In the context of RDF, related groupings result in subqueries with common subexpressions (graph patterns with overlapping structure) enabling opportunities for shared execution. For example, if two graph patterns in a query have the same structure (same join expression), then the graph pattern can be evaluated only once. In cases where graph patterns have subsumption relationship in join expressions, there may be opportunities to rewrite the query in a way that allows shared execution of common substructures. Even with structurally different graph patterns, there may be sharing opportunities within a MapReduce cycle.

Several techniques have been proposed to enable sharing of scans and computations across a MapReduce workload in order to reduce the associated I/O and network transfer costs, e.g., MRShare [30] proposes sharing of input scans, sharing map functions and map output, while executing a batch of grouping queries on a common input table. YSmart [28] groups correlated operations in complex queries, e.g., `Join`s and `GROUP BY`s accessing the same table, into a single MapReduce job to reduce redundant scans, computations, and network transfers (integrated into Hive 0.12.0).

A previous work [27] on multi-query optimization (MQO) of SPARQL queries, rewrites the input graph pattern queries into a set of queries Q_{OPT} using the SPARQL `OPTIONAL`[4] clause. Given a set of graph pattern queries Q with common substructures, the basic idea of SPARQL MQO is to (i) rewrite the input queries into a set of queries Q_{OPT} with `OPTIONAL` clauses (representing non-overlapping structures), (ii) evaluate queries Q_{OPT} over the RDF graph, and (iii) distribute the results of Q_{OPT} to input queries in Q. For example, two queries with the following set of triple patterns:

Q_1:`(?s p1 ?o)`

Q_2:`(?s1 p1 ?o1)(?s1 p2 ?o2)`

can be expressed using the `OPTIONAL` clause as follows:

Q_{OPT}:`(?s p1 ?o) OPTIONAL(?s p2 o2)`

where the non-overlapping triple pattern in Q_2 is specified as optional, i.e., resulting tuples may have NULL values for bindings of second triple pattern. Results matching original queries are extracted from results of Q_{OPT}. Note that multi-valued properties in the optional component may introduce duplicity and require special handling.

Discussion. A possible strategy to optimize RDF analytical queries is to rewrite and evaluate the individual graph patterns using the SPARQL MQO approach, extract answers to original graph patterns, and compute groupings over extracted subquery results. While such a rewriting seems beneficial when compared to sequential evaluation of individual graph patterns on MapReduce, our experiments on Hive showed that evaluating Q_{OPT} ahead of time prevents optimizations such as early projection and partial aggregations. This is because Q_{OPT} would need to be evaluated and stored as an intermediate table, since Hive neither supports logical views involving complex queries with multiple joins, nor does it support materialized views.

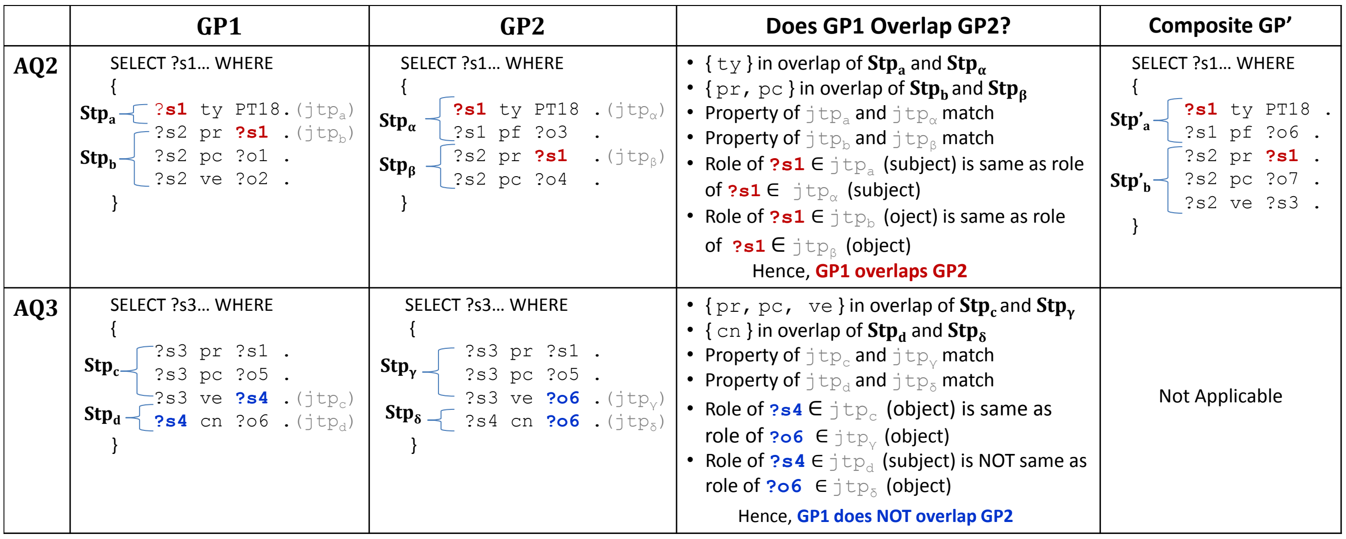
2.3 Rationale of Our Approach

We argue that it is necessary to approach the problem of optimizing RDF analytics *holistically*, rather than a two-step approach of independently optimizing the graph pattern matching phase and the grouping-aggregation phases. Given that RDF analytical queries often involve repeated computations over slightly different graph patterns, query plans that enable shared execution of common subpatterns are likely to compile into efficient execution plans. An important factor in this regard is the choice of algebra, the associated data model and the set of operators. One may use a relational-like algebra or alternatives such as the Nested TripleGroup Data Model and Algebra (NTGA) [33, 25]. We chose to use NTGA due to its underlying "groups of triples" or *triplegroup* model that enables concurrent computation of star-shaped join subpatterns (star-joins) in a query. The NTGA query plans not only enable sharing of scans and computations across multiple star subpatterns (resulting in shortened map-reduce execution workflows), but also concisely represent intermediate results in a denormalized form. In the next section, we build on the foundations of sharing that is already inherent in the NTGA approach and enhance its benefits by optimizing complex grouping-aggregation constraints.

[2]`https://pig.apache.org/`

[3]`https://hive.apache.org/`

[4]The `OPTIONAL` clause is used in SPARQL to allow querying of predicates that may not exist, i.e., answer is returned if there is a subgraph matching the `OPTIONAL` graph pattern, else it is ignored.



	GP1	GP2	Does GP1 Overlap GP2?	Composite GP'

AQ2	SELECT ?s1… WHERE
	{
	Stp$_a$ `?s1 ty PT18.(jtpa)`
	Stp$_b$ `?s2 pr ?s1 .(jtpb)`
	`?s2 pc ?o1 .`
	`?s2 ve ?o2 .`
	}
		SELECT ?s1… WHERE
		{
		Stp$_\alpha$ `?s1 ty PT18 .(jtpα)`
		`?s1 pf ?o3 .`
		Stp$_\beta$ `?s2 pr ?s1 .(jtpβ)`
		`?s2 pc ?o4 .`
		}
			• { `ty` } in overlap of **Stp$_a$** and **Stp$_\alpha$**
			• { `pr, pc` } in overlap of **Stp$_b$** and **Stp$_\beta$**
			• Property of `jtpa` and `jtpα` match
			• Property of `jtpb` and `jtpβ` match
			• Role of **`?s1`** ∈ `jtpa` (subject) is same as role of **`?s1`** ∈ `jtpα` (subject)
			• Role of **`?s1`** ∈ `jtpb` (oject) is same as role of **`?s1`** ∈ `jtpβ` (object)
			Hence, **GP1 overlaps GP2**
				SELECT ?s1… WHERE
				{
				Stp'$_a$ `?s1 ty PT18 .`
				`?s1 pf ?o6 .`
				Stp'$_b$ `?s2 pr ?s1 .`
				`?s2 pc ?o7 .`
				`?s2 ve ?s3 .`
				}

AQ3	SELECT ?s3… WHERE
	{
	Stp$_c$ `?s3 pr ?s1 .`
	`?s3 pc ?o5 .`
	`?s3 ve ?s4 .(jtpc)`
	Stp$_d$ `?s4 cn ?o6 .(jtpd)`
	}
		SELECT ?s3… WHERE
		{
		Stp$_\gamma$ `?s3 pr ?s1 .`
		`?s3 pc ?o5 .`
		`?s3 ve ?o6 .(jtpγ)`
		Stp$_\delta$ `?s4 cn ?o6 .(jtpδ)`
		}
			• { `pr, pc, ve` } in overlap of **Stp$_c$** and **Stp$_\gamma$**
			• { `cn` } in overlap of **Stp$_d$** and **Stp$_\delta$**
			• Property of `jtpc` and `jtpγ` match
			• Property of `jtpd` and `jtpδ` match
			• Role of **`?s4`** ∈ `jtpc` (object) is same as role of **`?o6`** ∈ `jtpγ` (object)
			• Role of **`?s4`** ∈ `jtpd` (subject) is NOT same as role of **`?o6`** ∈ `jtpδ` (object)
			Hence, **GP1 does NOT overlap GP2**
				Not Applicable

Figure 3: Structural overlap in graph patterns

Table 1: Quick Reference

Symbol	Description
tp	Triple pattern
jtp_i	Joining triple pattern in Stp_i
jv_{ij}	Variable joining tp_i and tp_j
GP	Graph pattern
Stp	Subject-rooted star subpattern
Stp_{abc}	Star pattern with property-set { a, b, c }
$Stp_{ab\underline{c}}$	Star pattern with primary properties a and b, and secondary (optional) property c
P_{prim}	Set of primary properties
P_{sec}	Set of secondary properties
tg	Triplegroup
TG	Set of triplegroups
TG_{abc}	Set of triplegroups with property-set { a, b, c }

Function	Returns
var(tp)	Set of variables in triple pattern tp
role($?v$)	Role of variable $?v$ (subject, property, or object)
prop(tp)	Property of triple pattern tp
props(Stp_i)	Set of properties in Stp_i
$\delta(?v)$	Variable substitution in a triple matching tp

3. ALGEBRAIC REWRITING OF SPARQL ANALYTICAL QUERIES

We reformulate SPARQL analytical queries with multiple grouping-aggregation constraints by, (i) identifying *overlap*s between graph patterns in a query based on structural constraints, (ii) evaluating a *composite graph pattern* that retrieves answers for original graph patterns, and (iii) computing required groupings and aggregations based on the composite graph pattern. Common notations and convenience functions used in this paper are summarized in Table 1.

Definition 3.1 ***(Overlapping Star Patterns)*** *Let Stp_1 and Stp_2 be two subject-rooted star subpatterns and let L be the intersection of their property sets, i.e., L = props(Stp_1) ∩ props(Stp_2). Then, Stp_1 and Stp_2 are considered to overlap if the following holds:*

- *Intersection of their property sets is non-empty, i.e., $L \neq \emptyset$.*
- *For any triple pattern tp_1 = ($s1$, rdf:type, $o1$) ∈ Stp_1, there exists some tp_2 = ($s2$, rdf:type, $o2$) ∈ Stp_2, with the same object component, i.e., $o1$ = $o2$.*

Figure 3 represents two analytical queries $AQ2$ and $AQ3$, each consisting of two graph patterns GP1 and GP2 (properties abbreviated). In the case of query $AQ2$, star pattern Stp$_a$ ∈ GP1 overlaps with Stp$_\alpha$ ∈ GP2 since both match on the object of **rdf : type** triple. Similarly, star patterns Stp$_b$ and Stp$_\beta$ overlap. The graph patterns in $AQ4$ also have two overlapping star patterns, i.e., Stp$_c$ structurally overlaps with Stp$_\gamma$, and Stp$_d$ overlaps with Stp$_\delta$.

Additionally, analytical queries may contain `FILTER` clauses that need to be considered while determining overlap between star patterns. For example, consider a filter on GP1 to retrieve a subset of products with price (property abbreviated as pc) > 5000, i.e., `FILTER`(?o5 > 5000). A possible strategy is to compute generalized composite star patterns (without filter) and apply restrictions prior to the aggregation phase. Pushing the filter to a later phase in the workflow may have implications on I/O and network transfer costs associated with materialization of some irrelevant intermediate results. Another interesting case is that of unbound-property star patterns containing triple patterns such as (`?s1 ?p o1`), used to query unknown or don't care relationships. Such queries need special handling, specifically if the unbound-property triple pattern participates in a join with other star patterns. Advanced optimizations for both these cases are out of scope of this paper. For the rest of this paper, we consider optimization of multi-graph-pattern queries involving bound-property star patterns with same filter constraints or filter constraints on a non-intersecting property.

Next, we generalize the notion of overlap to graph patterns by capturing similarity of join structures between star patterns. In order to do so, we introduce the concept of *role-equivalence* of join variables. Given two triple patterns tp$_1$ and tp$_2$, a join variable jv$_1$ is a variable in var(tp$_1$) ∩ var(tp$_2$). A join variable jv$_1$ ∈ tp$_1$ is said to be *role-equivalent* to join variable jv$_3$ ∈ tp$_3$ if, (i) the corresponding triple patterns agree on the property component, i.e., prop(tp$_1$) = prop(tp$_3$), and (ii) the join variables play the same role (subject, property, or object), i.e., role(jv$_1$) in tp$_1$ is the same as role(jv$_3$) in tp$_3$.

Definition 3.2 ***(Overlapping Graph Patterns)*** *Let graph pattern $GP1$ involve star subpatterns Stp_a, Stp_b,..., such that jv_{ab} de-*

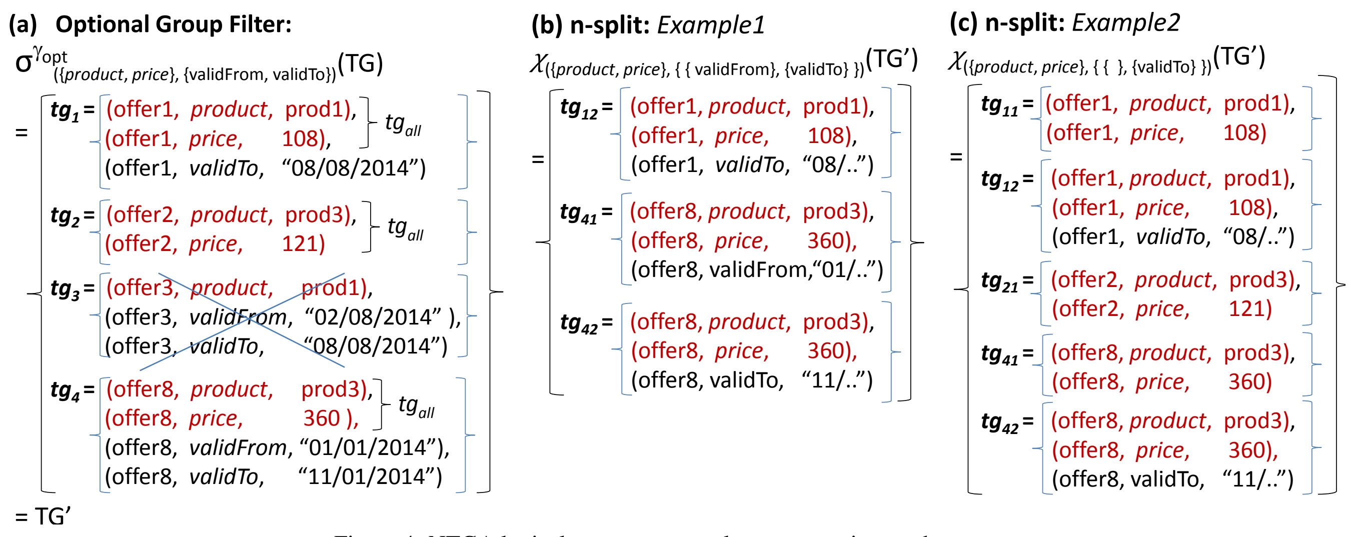

(a) Optional Group Filter:
σ^γopt ({product, price}, {validFrom, validTo})(TG)
tg1 = (offer1, product, prod1), (offer1, price, 108), (offer1, validTo, "08/08/2014")
tg_all
tg2 = (offer2, product, prod3), (offer2, price, 121)
tg3 = (offer3, product, prod1), (offer3, validFrom, "02/08/2014"), (offer3, validTo, "08/08/2014")
tg4 = (offer8, product, prod3), (offer8, price, 360), (offer8, validFrom, "01/01/2014"), (offer8, validTo, "11/01/2014")
= TG'
(b) n-split: Example1
χ({product, price}, { { validFrom}, {validTo} })(TG')
tg12 = (offer1, product, prod1), (offer1, price, 108), (offer1, validTo, "08/..")
tg41 = (offer8, product, prod3), (offer8, price, 360), (offer8, validFrom, "01/..")
tg42 = (offer8, product, prod3), (offer8, price, 360), (offer8, validTo, "11/..")
(c) n-split: Example2
χ({product, price}, { { }, {validTo} })(TG')
tg11 = (offer1, product, prod1), (offer1, price, 108)
tg12 = (offer1, product, prod1), (offer1, price, 108), (offer1, validTo, "08/..")
tg21 = (offer2, product, prod3), (offer2, price, 121)
tg41 = (offer8, product, prod3), (offer8, price, 360)
tg42 = (offer8, product, prod3), (offer8, price, 360), (offer8, validTo, "11/..")

Figure 4: NTGA logical operators to evaluate composite graph patterns

notes the variable that joins a triple pattern $jtp_a \in Stp_a$ with $jtp_b \in Stp_b$. Let graph pattern $GP2$ involve star subpatterns Stp_α, Stp_β,.... such that $jv_{\alpha\beta}$ denotes the variable that joins a triple pattern $jtp_\alpha \in Stp_\alpha$ with $jtp_\beta \in Stp_\beta$. Then, the graph patterns $GP1$ and $GP2$ are said to overlap if the following conditions hold:

- *Each star pattern $Stp_a \in GP1$ overlaps with some star pattern $Stp_\alpha \in GP2$*
- *Given a pair of overlapping star patterns Stp_a and Stp_α, their join variables jv_{ab} and $jv_{\alpha\beta}$ are role-equivalent.*

In the case of $AQ2$, graph patterns GP1 and GP2 overlap since both star patterns overlap and have the same join structure, e.g., subject-object join between $\mathtt{Stp_a}$ and $\mathtt{Stp_b}$ in GP1 matches the join structure between $\mathtt{Stp}_\alpha$ and $\mathtt{Stp}_\beta$ in GP2. In the case of $AQ3$, both star patterns overlap. However, $\mathtt{Stp_c}$ joins $\mathtt{Stp_d}$ using an object-subject join, where as $\mathtt{Stp}_\gamma$ joins $\mathtt{Stp}_\delta$ using an object-object join. Since the join structures are not similar, we consider GP1 and GP2 to be non-overlapping. Though there may be possibilities to share some scans and computations across non-overlapping graph patterns, for the rest of the paper we consider optimization of overlapping graph patterns.

Construction of a Composite Graph Pattern. Overlapping graph patterns GP1 and GP2 can be re-written as a *composite graph pattern* GP$'$ that captures the (non) overlapping substructures. For a pair of overlapping star patterns $Stp_a \in GP1$ and $Stp_\alpha \in GP2$, we define a composite star pattern Stp'_i such that:

- $\mathtt{props}(Stp'_i) = P_{prim} \cup P_{sec}$
- $P_{prim} = \mathtt{props}(Stp_a) \cap \mathtt{props}(Stp_\alpha)$, set of *primary* properties defining common substructures across star patterns.
- $P_{sec} = \{\ p_i \mid p_i \in \mathtt{props}(Stp_a) \cup \mathtt{props}(Stp_\alpha),\ p_i \notin P_{prim}\}$, set of *secondary* properties defining non-overlapping structures.

For example, $\mathtt{Stp_a} \in$ GP1 and $\mathtt{Stp}_\alpha \in$ GP2 can be rewritten as $\mathtt{Stp'_a}$ such that $\mathtt{props}(\mathtt{Stp'_a})$ = { ty18, $\underline{\mathtt{pf}}$ }, where ty18 (short for rdf : type PT18) is the primary property and pf is the secondary property (underlined). Similarly, $\mathtt{Stp_b} \in$ GP1 and $\mathtt{Stp}_\beta \in$ GP2 can be expressed as $\mathtt{Stp'_b}$ with set of properties { pr, pc, $\underline{\mathtt{ve}}$}. Query $AQ1$ can be re-written using a composite graph pattern:

$$\mathtt{GP'} = (\mathtt{Stp'_1} \bowtie \mathtt{Stp'_2} \bowtie \mathtt{Stp'_3})$$

where $\mathtt{props}(\mathtt{Stp'_1})$ = { ty18, $\underline{\mathtt{pf}}$ }, $\mathtt{props}(\mathtt{Stp'_2})$ = { pr, pc, ve}, and $\mathtt{props}(\mathtt{Stp'_3})$ = { cn }.

Answers matching a composite graph pattern may contain superfluous subtuples that do not match either of the original patterns, resulting in wrong aggregates. Hence, we need a way to validate join combinations.

An NTGA-based rewriting of a SPARQL analytical query requires support to compute and manipulate triplegroups that match composite star patterns and composite graph patterns. Specifically, we need support for the following operations – (i) A specialized triplegroup-filter operator that validates secondary (optional) properties in a composite star pattern; (ii) An operator to extract subsets of a triplegroup that match n original star patterns; (iii) A special join operator that restricts joins on valid combinations of composite star patterns; (iv) An operator in the spirit of *MD-Join* to compute grouping-aggregations on triplegroups. Next, we formally define the triplegroup-based logical operators. We assume our input to be a set of subject triplegroups (triples grouped on subject column).

3.1 Logical Operators

Definition 3.3 ***(Optional Group Filter)*** *Given a set of subject triplegroups TG and a star pattern Stp containing a set of primary properties P_{prim}, and a set of optional properties P_{opt}, the* ***optional group-filter*** *operator $\sigma^{\gamma opt}$ returns the subset of triplegroups in TG that contains a non-empty subset of triples matching all properties in P_{prim} and may contain triples matching properties in P_{opt}. Specifically,*

$$\sigma^{\gamma opt}_{(P_{prim},P_{opt})}(TG) := \{\ tg_i \in TG \mid P_{prim} \subseteq \mathtt{props}(tg_i) \subseteq (P_{prim} \cup P_{opt})\ \}$$

where $\mathtt{props}(tg_i)$ returns the set of properties in a triplegroup tg_i. Essentially, $\sigma^{\gamma opt}$ ensures that triplegroups contain a matching triple for each of the primary properties and may contain matches for properties in P_{opt}. For example, given $\mathtt{P_{prim}} = \{\mathtt{product}, \mathtt{price}\}$, triplegroup $\mathtt{tg_1}$, $\mathtt{tg_2}$, and $\mathtt{tg_4}$ are valid results for the $\sigma^{\gamma opt}$ expression in Figure 4(a). However, $\mathtt{tg_3}$ does not contain a matching triple for the primary property price, and hence gets filtered out. Note that valid triplegroups may have triples matching zero or more of the two optional properties $\mathtt{P_{opt}} = \{\mathtt{validFrom}, \mathtt{validTo}\}$.

Table 2: Evaluating composite graph patterns using α-Join

GP1	**GP2**	**GP'**	$\bowtie^{\gamma}_{(\alpha_1 \vee \alpha_2)}$**(...)**	
Stp_1:Stp_2	Stp_1:Stp_2	Stp'_1:Stp'_2	α_1	α_2
ab:de	ab:de	ab:de	–	–
ab:de	ab:def	ab:de$\underline{\text{f}}$	f $=\emptyset$	f$\neq\emptyset$
ab:de	abc:def	ab$\underline{\text{c}}$:de$\underline{\text{f}}$	c $=\emptyset \wedge$ f$=\emptyset$	c$\neq\emptyset \wedge$ f$\neq\emptyset$
abc:de	ab:def	ab$\underline{\text{c}}$:de$\underline{\text{f}}$	c$\neq\emptyset \wedge$ f$=\emptyset$	c $=\emptyset \wedge$ f$\neq\emptyset$
abc:de	ab:defg	ab$\underline{\text{c}}$:de$\underline{\text{fg}}$	c$\neq\emptyset \wedge$ f$=\emptyset$ $\wedge$ g$=\emptyset$	c $=\emptyset \wedge$ f$\neq\emptyset$ $\wedge$ g$\neq\emptyset$

Definition 3.4 ***(n-split)*** *Given a set of triplegroups TG, a set of primary properties P_{prim}, and n sets of secondary properties $\{P_{sec_1}$, P_{sec_2},..., $P_{sec_n}\}$, the **n-split** operator χ creates a set of n triplegroups as follows:*

$$\chi_{(P_{prim},\{P_{sec_1},P_{sec_2},\ldots,P_{sec_n}\})}(TG) := \{\ tg'_i,\ i \in [1,n]\}$$

such that:

- *$tg'_i = tg_{prim} \cup tg_{sec_i}$, where $tg_{prim}, tg_{sec_i} \subseteq tg$, $tg \in TG$*
- *$props(tg_{prim}) = P_{prim}$ and $props(tg_{sec_i}) = P_{sec_i}$*

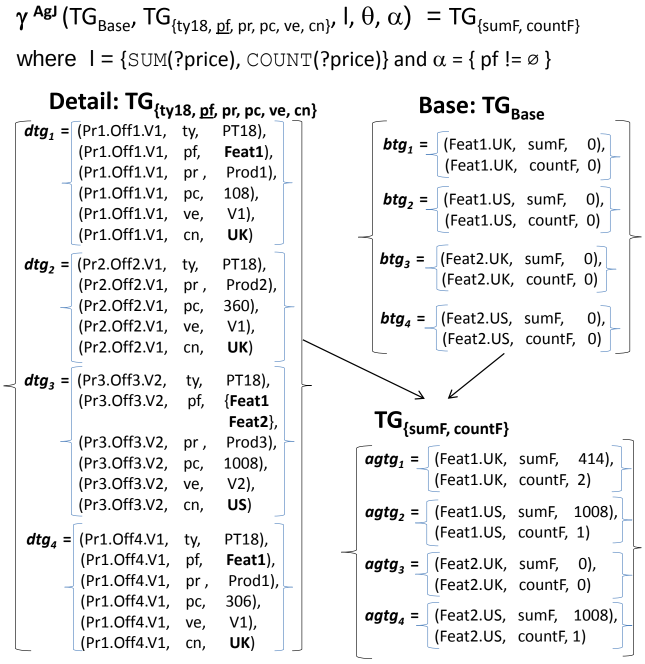
The n-split operator extracts n subsets of a triplegroup based on n sets of secondary properties, one for each of the original star patterns. Figure 4(b) shows triplegroups resulting from an n-split operation on $\mathtt{TG}'$ (n=2), with $\mathtt{P_{prim}} = \{\mathtt{product}, \mathtt{price}\}$, and two sets of secondary properties – $\mathtt{P_{sec_1}} = \{\mathtt{validFrom}\}$, and $\mathtt{P_{sec_2}} = \{\mathtt{validTo}\}$. While triplegroup $\mathtt{tg_{41}}$ conforms to the first pattern combination with properties { product, price, validFrom }, triplegroups $\mathtt{tg_{12}}$ and $\mathtt{tg_{42}}$ match the second combination { product, price, validTo}. Figure 4(c) shows another example of the n-split operation with $\mathtt{P_{sec_1}} = \{\}$ and $\mathtt{P_{sec_2}}$={ validTo }, i.e., the first combination contains only primary (no secondary) properties.

Let $\mathtt{GP_{abcde}}$ and $\mathtt{GP_{abdef}}$ be original graph patterns in a query and let $\mathtt{Stp_{ab\underline{c}}}$ and $\mathtt{Stp_{de\underline{f}}}$ be composite star patterns. The join ($\mathtt{Stp_{ab\underline{c}}} \bowtie \mathtt{Stp_{de\underline{f}}}$) may result in pattern combinations such as abde that do not match either of the original patterns and should be avoided. We encode valid pattern combinations using α conditions, a set of structural constraints on a TG equivalence class based on its secondary properties. For example, to ensure pattern combinations abcde, triplegroups in $\mathtt{TG_{ab\underline{c}}}$ must contain at least one triple with property c, represented as a constraint α: c $\neq \emptyset$, for brevity.

Definition 3.5 *(α**-Join**) Let TG_x and TG_y be two triplegroup equivalence classes that join on variables jv_x and jv_y belonging to joining triple patterns tp_x and tp_y, resp. Let α_1, α_2,...,α_m be m conditions involving secondary properties in the equivalence classes. Then the α**-Join** operator $\bowtie^{\gamma}_{\{\alpha_1 \vee \ldots \vee \alpha_m\}}$ creates a joined triplegroup involving $tg_x \in TG_x$ and $tg_y \in TG_y$ if the following holds:*

- *Triplegroup tg_x contains a matching triple for tp_x, and triplegroup tg_y contains a matching triple for tp_y, such that their variable substitutions match.*
- *tg_x and tg_y satisfy at least one of the α conditions.*

Table 2 shows examples of graph patterns GP1 and GP2, their composite graph pattern $\mathtt{GP}'$, and α constraints for the α-Join operator. For example, conditions α_1 and α_2 in row (5) correspond to the original graph patterns abcde and abdefg respectively, hence avoiding materialization of triplegroups matching irrelevant patterns such as abde, abdef, abdeg, abcdef, abcdefg, etc.



γ AgJ (TG_Base, TG_{ty18, pf, pr, pc, ve, cn}, l, θ, α) = TG_{sumF, countF}
where l = {SUM(?price), COUNT(?price)} and α = { pf != ∅ }
Detail: TG_{ty18, pf, pr, pc, ve, cn}
dtg1 = (Pr1.Off1.V1, ty, PT18), (Pr1.Off1.V1, pf, Feat1), (Pr1.Off1.V1, pr , Prod1), (Pr1.Off1.V1, pc, 108), (Pr1.Off1.V1, ve, V1), (Pr1.Off1.V1, cn, UK)
dtg2 = (Pr2.Off2.V1, ty, PT18), (Pr2.Off2.V1, pr , Prod2), (Pr2.Off2.V1, pc, 360), (Pr2.Off2.V1, ve, V1), (Pr2.Off2.V1, cn, UK)
dtg3 = (Pr3.Off3.V2, ty, PT18), (Pr3.Off3.V2, pf, {Feat1 Feat2}, (Pr3.Off3.V2, pr , Prod3), (Pr3.Off3.V2, pc, 1008), (Pr3.Off3.V2, ve, V2), (Pr3.Off3.V2, cn, US)
dtg4 = (Pr1.Off4.V1, ty, PT18), (Pr1.Off4.V1, pf, Feat1), (Pr1.Off4.V1, pr , Prod1), (Pr1.Off4.V1, pc, 306), (Pr1.Off4.V1, ve, V1), (Pr1.Off4.V1, cn, UK)
Base: TG_Base
btg1 = (Feat1.UK, sumF, 0), (Feat1.UK, countF, 0)
btg2 = (Feat1.US, sumF, 0), (Feat1.US, countF, 0)
btg3 = (Feat2.UK, sumF, 0), (Feat2.UK, countF, 0)
btg4 = (Feat2.US, sumF, 0), (Feat2.US, countF, 0)
TG_{sumF, countF}
agtg1 = (Feat1.UK, sumF, 414), (Feat1.UK, countF, 2)
agtg2 = (Feat1.US, sumF, 1008), (Feat1.US, countF, 1)
agtg3 = (Feat2.UK, sumF, 0), (Feat2.UK, countF, 0)
agtg4 = (Feat2.US, sumF, 1008), (Feat2.US, countF, 1)

Figure 5: Example Triplegroup *Agg-Join* operation that computes groupings based on `feature-country` combination

Definition 3.6 ***(TG Agg-Join)*** *Let TG_{base} and TG_{detail} be two triplegroup equivalence classes, θ be a condition involving variable substitutions in TG_{base} and TG_{detail}, and let l be a list of aggregation functions (f_1, f_2,...,f_m) over aggregation variables a_1, a_2, ..., a_m, respectively. Let α be a condition involving one of the secondary properties in TG_{detail}. Then the triplegroup Agg-Join operator,*

$$\gamma^{AgJ}(\ TG_{base},\ TG_{detail},\ l,\ \theta,\ \alpha)$$

creates a set of aggregated triplegroups ATG, where any aggregated triplegroup $agtg_i \in ATG$ satisfies the following conditions:

- *Each base triplegroup $btg_i \in TG_{base}$ is associated with a set of triplegroups in TG_{detail}, using the following function :*

  $$RNG(btg_i,\ TG_{detail},\ \theta,\ \alpha) = \{\ dtg \in TG_{detail}\ \}$$

  such that triplegroups btg_i and dtg satisfy conditions in θ and α.

- *Then, for each base triplegroup $btg_i \in TG_{base}$, an aggregated triplegroup $agtg_i \in ATG$ is produced with triples $t_{ik} \in agtg_i$ that contain values corresponding to some aggregation function f_k and variable a_k such that :*

  $$t_{ik} = (grpKey,\ createProp(f_k, a_k),\ f_k\_agtg_i\_a_k)$$

  whose values are computed as follows :

  – *grpKey is the subject of btg_i; createProp(f_k, a_k) returns a unique property based on combination of aggregation function and variable.*

  – *Aggregate $f_k\_agtg_i\_a_k$ is computed by applying the function f_k on variable substitutions of a_k in triplegroups matching $RNG(btg_i,\ TG_{detail},\ \theta,\ \alpha)$.*

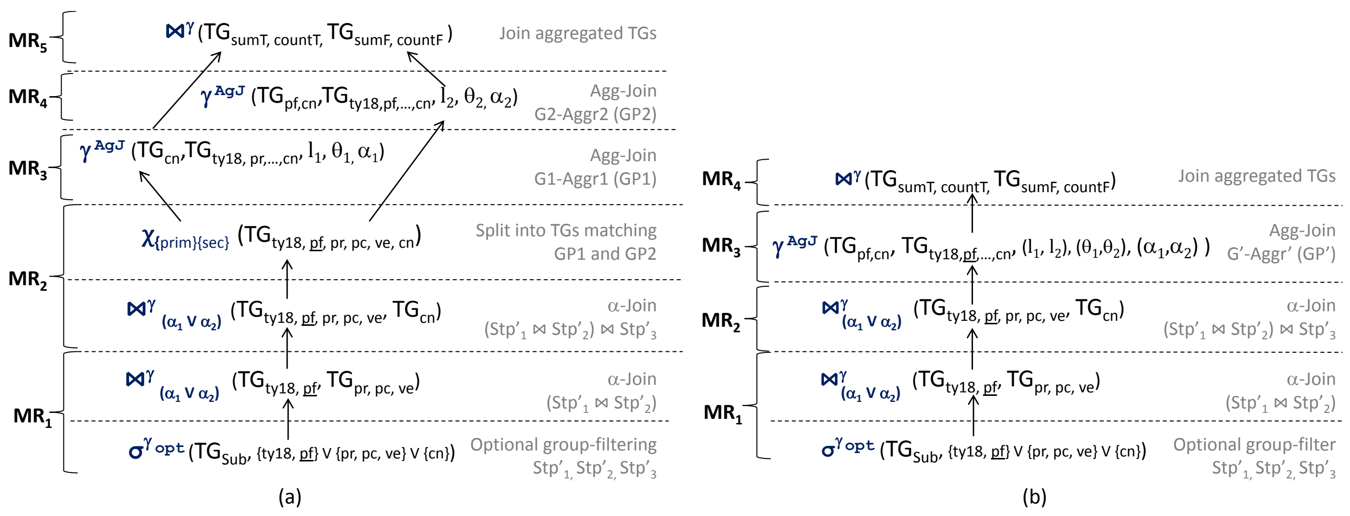

MR5
⋈γ (TGsumT, countT, TGsumF, countF)
Join aggregated TGs
MR4
γAgJ (TGpf,cn, TGty18,pf,...,cn, l2, θ2, α2)
Agg-Join G2-Aggr2 (GP2)
MR3
γAgJ (TGcn, TGty18, pr,...,cn, l1, θ1, α1)
Agg-Join G1-Aggr1 (GP1)
MR2
χ{prim}{sec} (TGty18, pf, pr, pc, ve, cn)
Split into TGs matching GP1 and GP2
⋈γ (α1 ∨ α2) (TGty18, pf, pr, pc, ve, TGcn)
α-Join (Stp'1 ⋈ Stp'2) ⋈ Stp'3
MR1
⋈γ (α1 ∨ α2) (TGty18, pf, TGpr, pc, ve)
α-Join (Stp'1 ⋈ Stp'2)
σγ opt (TGSub, {ty18, pf} ∨ {pr, pc, ve} ∨ {cn})
Optional group-filtering Stp'1, Stp'2, Stp'3
(a)
MR4
⋈γ (TGsumT, countT, TGsumF, countF)
Join aggregated TGs
MR3
γAgJ (TGpf,cn, TGty18,pf,...,cn, (l1, l2), (θ1,θ2), (α1,α2))
Agg-Join G'-Aggr' (GP')
MR2
⋈γ (α1 ∨ α2) (TGty18, pf, pr, pc, ve, TGcn)
α-Join (Stp'1 ⋈ Stp'2) ⋈ Stp'3
MR1
⋈γ (α1 ∨ α2) (TGty18, pf, TGpr, pc, ve)
α-Join (Stp'1 ⋈ Stp'2)
σγ opt (TGSub, {ty18, pf} ∨ {pr, pc, ve} ∨ {cn})
Optional group-filter Stp'1, Stp'2, Stp'3
(b)

Figure 6: Translation to MapReduce execution workflows: (a) Sequential and (b) Parallel evaluation of aggregations on a composite graph pattern GP'. Properties: `ty18` (`rdf : type PT18`), `pf` (`productFeature`), `pr` (`product`), `pc` (`price`), `ve` (`vendor`), `cn` (`country`)

A base triplegroup $btg_i \in TG_{base}$ corresponds to a distinct grouping key and produces an aggregated triplegroup $agtg_i \in ATG$. Subset of triplegroups in TG_{detail} that contribute to an aggregated triplegroup $agtg_i$ is computed using function $RNG(btg_i, TG_{detail}, \theta, \alpha)$, that returns the set of triplegroups in TG_{detail} that satisfy the join condition θ as well as the α condition with respect to the base triplegroup btg_i. The α condition defines restrictions based on secondary properties in TG_{detail}.

Figure 5 illustrates an example TG *Agg-Join* operation between TG equivalence classes $\texttt{TG}_{\texttt{Base}}$ (base) and $\texttt{TG}_{\{ty18,\underline{pf},pr,pc,ve,cn\}}$ (detail), to compute groupings based on feature and country. The RNG of a base triplegroup is calculated based on value bindings of the grouping variables ?`feature` and ?`country` in detail triplegroups (encoded as join condition θ). For triplegroup $\texttt{dtg}_1$, bindings δ_1(?`feature`)={ `Feat1` } and δ_1(?`country`)={ `UK` }. The α condition `pf`$\neq\emptyset$ ensures the presence of the secondary property `pf` (product feature). Triplegroup $\texttt{dtg}_2$ does not satisfy the α condition and hence does not contribute to any of the aggregated triplegroups. The RNG of base triplegroups is as follows:

$$RNG(\texttt{btg}_1, \texttt{TG}_{\{ty18,\underline{pf},pr,pc,ve,cn\}}, \theta, \alpha) = \{\ \texttt{dtg}_1, \texttt{dtg}_4\}$$
$$RNG(\texttt{btg}_2, \texttt{TG}_{\{ty18,\underline{pf},pr,pc,ve,cn\}}, \theta, \alpha) = \{\ \texttt{dtg}_3\ \}$$
$$RNG(\texttt{btg}_3, \texttt{TG}_{\{ty18,\underline{pf},pr,pc,ve,cn\}}, \theta, \alpha) = \emptyset$$
$$RNG(\texttt{btg}_4, \texttt{TG}_{\{ty18,\underline{pf},pr,pc,ve,cn\}}, \theta, \alpha) = \{\ \texttt{dtg}_3\ \}$$

Given a base triplegroup btg_i, the aggregated triplegroup is computed by aggregating triplegroups in RNG of btg_i. For example, $\texttt{agtg}_1$ is an aggregation of triplegroups $\texttt{dtg}_1$ and $\texttt{dtg}_4$ (RNG of $\texttt{btg}_1$). Note that RNG of $\texttt{btg}_3$ is empty and the aggregated triplegroup $\texttt{agtg}_3$ retains default values.

4. QUERY EXECUTION ON MAPREDUCE

In MapReduce, data processing tasks (or queries) are encoded as a sequence of map-reduce function pairs which are executed in parallel on a cluster of machines. Extended MapReduce systems such as Apache Hive and Pig support high-level query primitives that are automatically compiled into a MapReduce execution workflow. The proposed logical operators were integrated into an NTGA-based extension of Apache Pig, called *RAPID+* [33, 25]. The extended system, called *RAPIDAnalytics*, includes proposed optimizations to evaluate multi-aggregation SPARQL analytical queries. Both systems parse graph pattern queries in SPARQL and support a set of logical and physical operators for both Pig and NTGA. Interested readers can refer to [25] for architectural details of *RAPID+*.

4.1 Translation to MapReduce Plans

As with other relational-style Hadoop extensions, query compilation process in *RAPIDAnalytics* begins with a logical plan, which is compiled into a physical plan with physical operators. A physical operator is either a single function or a function pair that corresponds to map and reduce phases of the logical operator. For example, the optional group-filtering operator `TG_OptGrpFilter` ($\sigma^{\gamma_{opt}}$) is a single function and can be pipelined with other operators in either the map or the reduce phases. However, operators such as the triplegroup Agg-Join `TG_AgJ` which require redistribution of input, are defined as map-reduce function pairs. The assignment of the physical operators to MapReduce cycles constitutes a MapReduce plan.

Next, we summarize the execution workflow of our example query $AQ1$ on MapReduce. As described earlier, overlapping graph patterns `GP1` and `GP2` are re-written as a composite graph pattern:

$$\texttt{GP}'\text{: } \texttt{Stp}_{ty18,\underline{pf}} \bowtie \texttt{Stp}_{pr,pc,ve} \bowtie \texttt{Stp}_{cn}$$

Let $\texttt{TG}_{Sub}$ be a set of subject triplegroups (set of triples grouped by Subject column). Figure 6(a) shows the query plan with the assignment of operators to map-reduce (MR) cycles. The optional group-filtering operator creates three sets of triplegroup equivalence classes – $\texttt{TG}_{\{ty18,\underline{pf}\}}$, $\texttt{TG}_{\{pr,pc,ve\}}$, and $\texttt{TG}_{\{cn\}}$, that match the composite star patterns. The two α-Join operators compute the α-join between triplegroups to compute matches to the composite graph pattern. The *n-split* operator extracts matches to the original graph patterns `GP1` and `GP2`. Subsequently, the two TG Agg-Join operators (γ^{AgJ}) compute the aggregations per country and per feature-country, resp. The final ratio is computed by joining the aggregated TG equivalence classes using a map-only phase.

An useful optimization [10, 5] is that a series of aggregations on the same detail relation can be evaluated in parallel if they are independent, i.e., the θ conditions of the second *Agg-Join* does not involve values generated by the first *Agg-Join*. Figure 6(b) shows the NTGA query plan and MapReduce execution plan that enables parallel execution of the TG *Agg-Join* operator by combining them as a generalized operator (executed in MR cycle MR_3):

$$\gamma^{AgJ}(\mathtt{TG}_{g1}, \mathtt{TG}_{\{ty18,\underline{pf},pr,pc,ne,cn\}}, (\mathtt{l}_1, \mathtt{l}_2), (\theta_1, \theta_2), (\alpha_1, \alpha_2))$$

4.2 Algorithms for Physical Operators

Algorithm 1 gives an overview of the job flow for key phases in *RAPIDAnalytics* – Job_i, that computes the join between the triplegroup equivalence classes, and Job_k, that computes the aggregate join between the triplegroup equivalence classes. If there is a structural overlap in the input graph patterns, the triplegroup equivalence classes are computed based on the *composite* graph pattern. This is achieved by evaluating the optional group-filtering operator, `TG_OptGrpFilter`, based on the required and optional properties in the composite graph pattern. Below are map-reduce algorithms for the physical operators.

Algorithm 1: MR job workflow in *RAPIDAnalytics*

```
//$Job_i$:$\alpha$-Join between TG equivalence classes
Map:
    TG' ← TG_OptGrpFilter(TG, <EC,{$P_{prim}$, $P_{opt}$}>);
    TG_AlphaJoin($TG'$).Map();
Reduce:
    TG" ← TG_AlphaJoin(TG').Reduce();
//$Job_k$:Agg-Join on TG equivalence classes
Map:
    TG_AgJ(TG").Map();
Reduce:
    AggTG ← TG_AgJ(TG").Reduce();
//$Job_n$:Join Aggregated TGs
Map:
    TG_Join(AggTG);
```

`TG_AlphaJoin`: The input to this operator is a set of annotated triplegroups (matching a composite subpattern) whose join is to be computed. In order to eliminate pattern combinations that do not match any of the original graph patterns, all valid combinations are encoded as a list of α conditions, one for each of the original graph patterns. Algorithm 2 shows the map-reduce functions for the `TG_AlphaJoin` operator that integrates α-based filtering of irrelevant triplegroups during the join between equivalence classes.

In the map phase, an input triplegroup is tagged either on the *Subject* or *Object* value, based on the type of join. Each *reduce*() receives annotated triplegroups corresponding to the same join key. The algorithm iterates through triplegroups in the left equivalence class (*leftEC*) and right equivalence class (*rightEC*), and computes the join only if at least one of the α conditions is satisfied. For example, two triplegroups with properties `ab` and `de`, are not joined if the valid pattern combinations are `abcde` and `abdef`.

`TG_AgJ`: The input to this operator is a set of annotated triplegroups that match the composite graph pattern. The output is a set of aggregated triplegroups that contain the required aggregations. Algorithm 3 shows the map-reduce functions for the `TG_AgJ` operator. In order to reduce the number of intermediate triplegroups that are shuffled to the reducers, we implement a hash-based aggregation per mapper, i.e., instead of generating map output for each map input triplegroup, we partially aggregate the triplegroups at each mapper. The triplegroups are aggregated into a hashmap *multiAggMap* that is accessible across different map() invocations at a mapper. This hash-based aggregation resembles a local combiner within each mapper.

Each *Agg-Join agj* (identified by *id*) contains a θ condition, from which the grouping key *grp* is extracted. In the map phase,

Algorithm 2: **`TG_AlphaJoin`** (Triplegroup α-Join)

```
Map (key:null, val: AnnTG atg)
    if join on Subj then
        emit ⟨ $atg.Sub$, $atg$ ⟩;
    else if join on Obj then
        objList ← extract objects corr. to join property from $atg$;
        foreach obj ∈ objList do
            emit ⟨ $obj$, $atg$ ⟩;
Reduce (key:joinKey, val:List of AnnTGs $TG'$) ;
    $\alpha List < \alpha_1, ..., \alpha_n > \leftarrow \alpha$ restrictions for current join;
    leftList ← extract leftEC AnnTGs from $TG'$;
    rightList ← extract rightEC AnnTGs from $TG'$;
    foreach $ltg$ ∈ leftList do
        foreach $rtg$ ∈ rightList do
            if $\exists\ \alpha \in \alpha List$ such that $ltg$ and $rtg$ satisfy $\alpha$ then
                emit ⟨ $joinTGs(ltg, rtg)$⟩;
```

as each input triplegroup atg is processed, aggregations are computed if the α condition is satisfied. Once all aggregations for agj are computed, triplegroup *currAggTg* is aggregated with existing values in the mapper's global hashmap *multiAggMap*. Once the map() functions are complete, pre-aggregated entries in the global hashmap *multiAggMap* are output. Each reduce() receives pre-aggregated triplegroups corresponding to the same *id-grp* combination and further aggregates them.

Algorithm 3: **`TG_AgJ`** (Triplegroup Agg-Join)

```
Map (k:null, v: AnnTG atg)
    //Initialize multiAggMap for Map()
    //aggregation
    foreach $agj < id, aggList, theta, alpha > \in agjList$ do
        if $atg$ satisfies $alpha$ then
            grp ← extract $agj.theta$ from $atg$ ;
            curAggTg ← Aggregate atg based on $aggList$;
            Aggregate curAggTg to multiAggMap(k:id#grp);
Map.clean ()
    Emit pre-aggregated entries in multiAggMap;
Reduce (k:id#grp, v:List of AggTGs TG) ;
    grpAggTg ← Aggregate TG based on aggList;
    Emit aggregated triplegroup grpAggTg;
```

5. EMPIRICAL EVALUATION

This section presents a comprehensive evaluation of the proposed algebraic optimizations for RDF analytical queries. The performance of *RAPIDAnalytics* with two Hive approaches, (i) *Hive (Naive)*, SPARQL query translated into HiveQL, and (ii) *Hive (MQO)*, an MQO-based rewriting [27] of graph patterns using left outer joins, followed by a second HiveQL query to compute associated grouping and aggregations. Evaluation also included *RAPID+ (Naive)* [25], NTGA-based sequential evaluation of multiple graph patterns and grouping-aggregation phases.

5.1 Setup

Experiments were conducted on NCSU's VCL [34], where each node in the cluster was a dual core Intel X86 machine with 2.33GHz processor speed, 4GB memory, running Red Hat Linux. 10, 50, and 60-node Hadoop clusters (block size 128MB, 1GB heap-size for child jvms) were used with Hive release 0.12.0 and Hadoop 0.20.2.

Testbed - Dataset and Queries. Two synthetic datasets were generated by the Berlin SPARQL Benchmark (BSBM) [1] data

Query	GP1*	Group BY	GP2*	Group BY
MG1:lo, MG2:hi	3:2	{feature}	2:2	ALL
MG3:lo, MG4:hi	3:3:1	{feature, country}	2:3:1	{country}
MG6	4:2:2	{cid, gene}	4:2:2	{cid}
MG7	4:2:2	{cid, drug}	4:2:2	{cid}
MG8	4:2:2	{cid, gene}	4:2:2	ALL
MG9	2:1	{gene}	2:1	ALL
MG10	3:1	{disease, gene}	2:1	{gene}
MG11	2:2	{country}	2:1	ALL
MG12	2:2	{country, pubType}	2:1	{country}
MG13	3:1	{author,pubType}	3:1	{pubType}
MG14	3:1	{author,pubType}	3:1	{pubType}
MG15:lo	3:1	{authorlastname}	3:1	ALL
MG16:hi	3:1	{authorlastname}	3:1	ALL
MG17	3:2	{country}	3:1	ALL
MG18	3:2	{author, country]	2:2	{country}
* No. of triple patterns in $Stp_1 : Stp_2 : \ldots$				

Figure 7: Evaluated RDF Analytical Queries

generator tool – *BSBM-500K* (43GB, 500K Products, ∼175M triples) and *BSBM-2M* (172GB, 2M Products, ∼700M triples). Evaluation of real-world RDF analytical queries was conducted on a chemogenomics RDF data warehouse, *Chem2Bio2RDF* [13], that is an aggregation of data from multiple chemical, biological, and chemogenomics data sources that link chemical compounds with targets, genes, side-effects, diseases, and publications (60GB, ∼340M triples). Additional experiments were conducted on a second real-world dataset, *PubMed* (*Bio2RDF* release 2) [8] (230GB, ∼1.7B triples).

The evaluation tested simple ($G1$-$G9$) as well as multi-grouping queries ($MG1$-$MG18$) with varying selectivities, varying granularity of groupings (`GROUP BY` ALL vs. `GROUP BY` feature), and varying structures of associated graph patterns, as summarized in Figure 7. Queries $G1$-$G4$ and $MG1$-$MG4$ were adapted from the BSBM Business Intelligence Use Case 3.1 [1], an e-commerce use case. Queries $G5$-$G9$ and $MG6$-$MG10$ were adapted based on case studies [13] on the Chem2Bio2RDF dataset, with use cases such as disease-specific drug discovery. Queries $MG11$-$MG18$ involve PubMed records. Additional details about all evaluated queries in SPARQL and Hive scripts are available on the project website [2].

Pre-processing. For Hive approaches, triples were vertically partitioned (VP) [3] and loaded into Hive tables with property-object partitions for *rdf:type* triples. All Hive tables were stored as *Optimized Row Columnar* (ORC)[5] file format which aggressively compresses data (∼80-96% reduction in data size with default compression) and has optimizations such as light-weight indexes to skip row groups for predicate-based filtering, column-level aggregates etc. For *RAPIDAnalytics* and *RAPID+*, triples were grouped on subject column to generate subject triplegroups, stored in text files based on equivalence class (set of properties). Further, *rdf:type* triples with ProductType objects were grouped based on prefixes to avoid creation of multiple small files. Additional details about the pre-processing phase is available on the project website [2].

Query	BSBM				Query	Chem2Bio2RDF	
	500K		2M			Hive	R.A.
	Hive	R.A.	Hive	R.A.			
G1:lo	1023	209	3261	215	**G5**	144	124
G2:hi	974	182	3002	158	**G6**	99	102
G3:lo	1632	287	6088	302	**G7**	105	118
G4:hi	1112	183	5419	170	**G8**	142	104
					G9	535	91

Table 3: Performance comparison of *Hive* and *RAPIDAnalytics* (R.A.) with varying structures of groupings (in seconds)

5.2 Evaluation Results

Varying Structure of Groupings. Four single-grouping queries were evaluated with varying selectivity of graph patterns and group granularity ($G1$-$G2$ with `GROUP BY` ALL and $G3$-$G4$ with `GROUP BY` feature). Queries $G1$ and $G3$ pertain to ProductType1 (low selectivity), while $G2$ and $G4$ pertain to ProductType9 (high selectivity). Table 3 shows a performance comparison of *Hive* and *RAPIDAnalytics* for *BSBM-500K* (10-node cluster). *Hive* requires 4 MR cycles for all queries (MR_1-MR_2 for star patterns, MR_3 to join the stars, and MR_4 to compute grouping-aggregation). In cases where (n-1) of the joining relations are small enough to fit in memory, *Hive* uses a map-join (map-only MR cycle), e.g., all subqueries involving ProductType1 and ProductType9. Also *Hive* enables optimizations such as push down of `PROJECT`s and partial aggregation during preceding join operations. *RAPIDAnalytics* executes all four queries in 2 cycles (MR_1 for graph pattern processing and MR_2 for the *Agg-Join* operation), with a consistent performance gain of ∼80% over *Hive* for all four queries.

Multiple Grouping-Aggregation Constraints. Figure 8(a-b) shows a performance comparison of all four approaches for queries $MG1$-$MG4$ with lo (low) and hi (high) query selectivity. Queries $MG1$-$MG2$ require 3 MR cycles per graph pattern in *Hive*, followed by 2 cycles for the grouping-aggregation (total 9 cycles). MQO-based *Hive* approach executes the composite graph pattern in 3 cycles, followed by 4 MR cycles to extract the distinct combinations matching the original patterns and compute the aggregations (total 7 cycles). *RAPID+* requires 2 MR cycles per subquery (1 MR for graph pattern matching, 1 MR for grouping-aggregation) and a map-only cycle to join the aggregated results (total 5 MR cycles). *RAPIDAnalytics* evaluated $MG1$-$MG2$ in 3 cycles (MR_1 to compute the composite graph pattern, MR_2 for parallel evaluation of the two grouping-aggregations and a map-only MR_3 to join the aggregated triplegroups).

Queries $MG3$-$MG4$ involve complex graph patterns with 3 star patterns. Sequential graph pattern processing in naive *Hive* results in a total of 11 MR cycles, while MQO-based *Hive* approach takes half the number of cycles for evaluating the composite graph pattern (8 MR cycles). *RAPID+* requires 2 MR cycles per graph pattern (7 MR cycles), while *RAPIDAnalytics* further reduces the number of cycles to 4 by parallel evaluation of the two grouping-aggregations. In general, the algebraic optimization in *RAPIDAnalytics* to group and aggregate on a composite graph pattern showed 30-45% gains over sequential evaluation of the different phases using naive *RAPID+*.

Scalability Study. Table 3 and Figure 8(b) show performance comparisons of 8 queries on a larger dataset *BSBM-2M*. The compression of input and intermediate results using the ORC File format, initializes less number of mappers (incur the overhead of decompression). *RAPID+* and *RAPIDAnalytics* initiate more number of mappers for most MR cycles leading to better utilization of re-

[5] `https://cwiki.apache.org/confluence/display/Hive/LanguageManual+ORC`

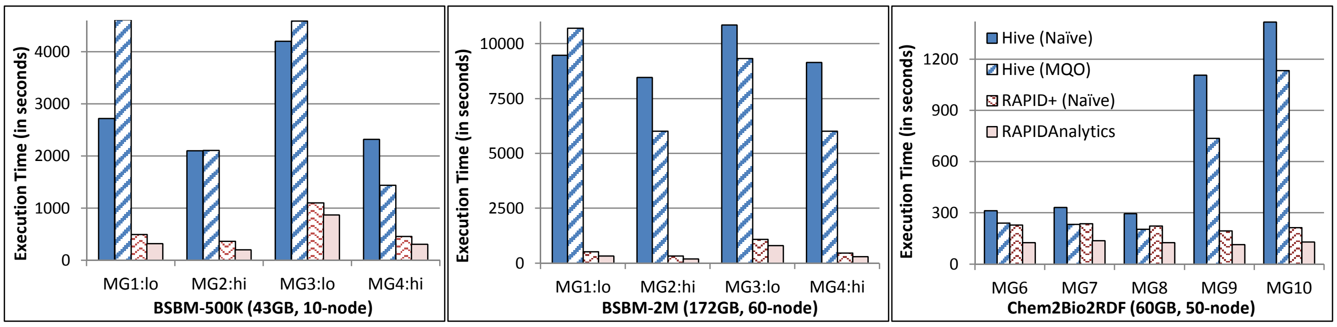


Figure 8: A performance comparison for multi-grouping SPARQL analytical queries

sources. For multi-grouping queries, *Hive (MQO)* did better than *Hive* for most cases with larger dataset due to higher savings in materialization of intermediate results, associated I/Os, and network transfers. *RAPIDAnalytics* showed 90-93% performance gains over *Hive (MQO)* for queries $MG1$-$MG2$ on *BSBM-500K*, which further increased to 97% with *BSBM-2M*. Similar increase was seen for queries $MG3$-$MG4$, where performance gains of *RAPIDAnalytics* over *Hive (MQO)* increased from 78-81% to 93% with the larger setup.

Real-world RDF Analytics. Table 3 shows results for queries $G5$-$G9$ on Chem2Bio2RDF. Query $G5$ with 6 join operations was evaluated by *Hive* using map-only joins (due to small size VP tables). Similar optimizations were enabled by *Hive* for $G6$-$G8$, with clear benefits seen in the case of $G7$, where *RAPIDAnalytics* takes 12 additional seconds when compared to *Hive*. Query $G9$ involves medline properties with large VP tables, forcing *Hive* to use full map-reduce cycles. *RAPIDAnalytics* shows 83% performance gain over *Hive* for $G9$. Figure 8(c) shows results for multi-aggregation queries, i.e., $MG6$-$MG8$ with high selectivity (small VP relations), while queries $MG9$-$MG10$ involve large VP relations. Naive *Hive* evaluates query $MG6$ using 13 MR cycles (11 map-only), while MQO-based *Hive* approach requires 8 MR cycles (6 map-only). *RAPID+* evaluates $MG6$ using 7 MR cycles (all map-reduce), with execution times almost comparable with *Hive (MQO)*. *RAPIDAnalytics* requires a total of 4 MR cycles. In general, even though the *Hive*-based approaches evaluate most of the joins in $MG6 - MG8$ as map-joins, *RAPIDAnalytics* shows a performance gain of 40-50% over *Hive (MQO)* and 60% gains over naive *Hive* for queries $MG6$-$MG8$. In case of queries $MG9$-$MG10$, the findings are similar to BSBM datasets, with *RAPIDAnalytics* showing close to 90% performance gain over Hive approaches.

Results for the *Pubmed* dataset are summarized in Table 4. Queries $MG11 - MG12$ and $MG17 - MG18$ compute groupings over PubMed records, the associated grants, and the countries where the grants are issued. Queries $MG13$-$MG16$ compute groupings based on publication type and authors of PubMed records and aggregate the number of Medical Subject (MeSH) Headings (query $MG13$) or associated chemicals (queries $MG14$-$MG16$). Further, selectivity of the queries were varied by querying different types of publications, e.g., $MG15$ and $MG16$ have similar query structure except that $MG15$ retrieves PubMed records with publication type "Journal Article" while $MG16$ concerns publications of type "News" (higher selectivity than journal article). Across all queries, *RAPIDAnalytics* showed improvements of above 93% over both Hive approaches. Hive performed the worst for queries

Query	**PubMed** (230GB dataset, 60-node cluster)			
	Hive (Naive)	**Hive (MQO)**	**RAPID+ (Naive)**	**RAPID Analytics**
MG11	2111	1753	229	124
MG12	2771	2898	229	126
MG13	120min*	15060	1102	651
MG14	18713	9124	756	462
MG15	13746	7320	619	338
MG16	10777	5795	464	237
MG17	2210	1851	226	118
MG18	5654	4817	306	202

* Eventually failed due to insufficient HDFS disk space.

Table 4: Evaluation of real-world queries on PubMed dataset (execution time in seconds)

$MG13$-$MG16$ that involve large VP relations (MeSH heading and chemical), due to the initiation of less number of mappers based on compressed (ORC) file sizes. Furthermore, while the Hive MQO approach eventually finished execution for query MG13, the naive Hive approach failed while computing the second graph pattern due to insufficient disk space. This is because one of the star-join cycles produces join output of size 190GB, which is materialized twice in the case of sequential execution of graph patterns, thus increasing the overall demand of required HDFS disk space. On the contrary, *RAPIDAnalytics* benefits from the concise representation of intermediate results using the NTGA approach while representing join results involving the multi-valued property MeSH heading. Further, the shared execution of graph patterns in *RAPIDAnalytics*, results in less number of materialization steps and less demand on required disk space. Overall, RAPIDAnalytics resulted in 40-48% performance gains over the sequential execution of graph patterns in RAPID+.

Discussion. Though *Hive(MQO)* compiles into a shorter execution workflow when compared to naive Hive, in some cases the performance is worse than sequential execution of subqueries. This is because of Hive's lack of support for materialized views or views with complex join expressions, forcing the evaluation of the composite graph pattern as a separate HiveQL query. A direct implication of this is that optimizations based on the final query such as early projections and partial aggregations, which reduce the I/O and materialization in the intermediate phases, are not applicable. Another observation is that vertical-partitioning coupled with the ORC file format can be beneficial for queries that involve high-selectivity properties. Irrespective of the selectivity of the involved properties, the algebraic optimization techniques in *RDFAnalytics* were found to be beneficial for multi-grouping queries by enabling shared execution of graph patterns as well as the required aggregations. *RAPIDAnalytics* can further benefit by integration of optimizations such as map-side joins and partial aggregations. While SPARQL analytical queries with unbound properties were not considered in this work, proposed optimizations in this paper can be extended based on NTGA-based optimizations in [32] to support composite graph patterns involving unbound-property triple patterns.

6. CONCLUSION AND FUTURE WORK

In this paper, we presented an algebraic optimization of SPARQL analytical queries that enables shared execution of common subexpressions across related groupings. Such a refactoring allows parallel evaluation of independent aggregations with savings in I/O and processing costs, a critical requirement while supporting large scale RDF analytics on Cloud platforms. Experiments on real-world and synthetic benchmark datasets showed promising results for SPARQL queries with multi-aggregation constraints. A natural extension of this work is to support more complex OLAP queries on RDF data models.

7. ACKNOWLEDGMENTS

The work presented in this paper is partially funded by NSF grant IIS-1218277.

8. REFERENCES

[1] BSBM Business Intelligence 3.1. `http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/spec/BusinessIntelligenceUseCase/`.

[2] Project Website: RAPIDAnalytics. `http://research.csc.ncsu.edu/coul/RAPID/RAPIDAnalytics`.

[3] D.J. Abadi, A. Marcus, S.R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB Endowment*, 2007.

[4] F.N. Afrati and J.D. Ullman. Optimizing joins in a map-reduce environment. In *ACM EDBT*, 2010.

[5] Michael O Akinde and Michael H Böhlen. Generalized md-joins: Evaluation and reduction to sql. In *Databases in Telecommunications II*. 2001.

[6] Michael O Akinde, Michael H Böhlen, Theodore Johnson, Laks VS Lakshmanan, and Divesh Srivastava. Efficient olap query processing in distributed data warehouses. *Information Systems*, 28(1), 2003.

[7] Jens Albrecht and Wolfgang Lehner. On-line analytical processing in distributed data warehouses. In *IEEE IDEAS*, 1998.

[8] François Belleau, Marc-Alexandre Nolin, Nicole Tourigny, Philippe Rigault, and Jean Morissette. Bio2rdf: towards a mashup to build bioinformatics knowledge systems. *Journal of biomedical informatics*, 41(5), 2008.

[9] Don Chamberlin. *Using the new DB2: IBM's object-relational database system*. 1996.

[10] D. Chatziantoniou, T. Johnson, M. Akinde, and S. Kim. The md-join: An operator for complex olap. In *IEEE ICDE*, 2001.

[11] Damianos Chatziantoniou and Elias Tzortzakakis. Asset queries: a declarative alternative to mapreduce. *ACM SIGMOD Record*, 38(2), 2009.

[12] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and olap technology. *ACM Sigmod record*, 26(1), 1997.

[13] Bin Chen, Xiao Dong, Dazhi Jiao, Huijun Wang, Qian Zhu, Ying Ding, and David J Wild. Chem2bio2rdf: a semantic framework for linking and data mining chemogenomic and systems chemical biology data. *BMC bioinformatics*, 11(1), 2010.

[14] Lei Chen, Christopher Olston, and Raghu Ramakrishnan. Parallel evaluation of composite aggregate queries. In *IEEE ICDE*, 2008.

[15] Dario Colazzo, François Goasdoué, Ioana Manolescu, and Alexandra Roatiş. RDF Analytics: Lenses over Semantic Graphs. In *Proc. WWW*, 2014.

[16] Richard Cyganiak, Dave Reynolds, and Jeni Tennison. The rdf data cube vocabulary. *W3C Recomm.*, 2013.

[17] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1), 2008.

[18] Lorena Etcheverry and Alejandro A Vaisman. Enhancing olap analysis with web cubes. In *The Semantic Web: Research and Applications*. 2012.

[19] Goetz Graefe, Usama M Fayyad, Surajit Chaudhuri, et al. On the efficient gathering of sufficient statistics for classification from large sql databases. In *KDD*, 1998.

[20] Jim Gray, Adam Bosworth, Andrew Layman, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In *ICDE*, 1996.

[21] Venky Harinarayan, Anand Rajaraman, and Jeffrey D Ullman. Implementing data cubes efficiently. *ACM SIGMOD Record*, 25(2), 1996.

[22] Steve Harris and Andy Seaborne. Sparql 1.1 query language. *W3C Recomm.*, 21, 2013.

[23] J. Huang, D.J. Abadi, and K. Ren. Scalable sparql querying of large rdf graphs. *VLDB Endowment*, 4(11), 2011.

[24] Benedikt Kampgen, Sean ORiain, and Andreas Harth. Interacting with statistical linked data via olap operations. In *Interacting with Linked Data*, 2012.

[25] H.S. Kim, P. Ravindra, and K. Anyanwu. From sparql to mapreduce: The journey using a nested triplegroup algebra. *VLDB Endowment*, 4(12), 2011.

[26] Hugo YK Lam, Luis Marenco, Tim Clark, et al. Alzpharm: integration of neurodegeneration data using rdf. *BMC bioinformatics*, 8(3), 2007.

[27] Wangchao Le, Anastasios Kementsietsidis, Songyun Duan, and Feifei Li. Scalable multi-query optimization for sparql. In *IEEE ICDE*, 2012.

[28] R. Lee, T. Luo, Y. Huai, F. Wang, Y. He, and X. Zhang. Ysmart: Yet another sql-to-mapreduce translator. In *IEEE ICDCS*, 2011.

[29] A. Nandi, C. Yu, P. Bohannon, and R. Ramakrishnan. Distributed cube materialization on holistic measures. In *IEEE ICDE*, 2011.

[30] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. Mrshare: Sharing across multiple queries in mapreduce. *VLDB Endowment*, 3(1-2), 2010.

[31] Patrick O'Neil and Dallan Quass. Improved query performance with variant indexes. *ACM Sigmod Record*, 26(2), 1997.

[32] P. Ravindra and K. Anyanwu. Scaling unbound-property queries on big RDF data warehouses using mapreduce. In *EDBT*, 2015.

[33] P. Ravindra, H.S. Kim, and K. Anyanwu. An intermediate algebra for optimizing rdf graph pattern matching on mapreduce. *The Semantic Web: Research and Applications*, 2011.

[34] H.E. Schaffer, S.F. Averitt, M.I. Hoit, A. Peeler, E.D. Sills, and M.A. Vouk. Ncsu's virtual computing lab: a cloud computing solution. *Computer*, 42(7), 2009.

[35] Ambuj Shatdal and Jeffrey F. Naughton. Adaptive parallel aggregation algorithms. In *ACM SIGMOD*, pages 104–114, 1995.

[36] R. Sridhar, P. Ravindra, and K. Anyanwu. Rapid: Enabling scalable ad-hoc analytics on the semantic web. *The Semantic Web-ISWC*, 2009.

[37] Ming-Chuan Wu and Alejandro P Buchmann. Encoded bitmap indexing for data warehouses. In *IEEE ICDE*, 1998.

[38] Amrapali Zaveri, Ricardo Pietrobon, Soren Auer, Jens Lehmann, Michael Martin, and Timofey Ermilov. Redd-observatory: Using the web of data for evaluating the research-disease disparity. In *IEEE/WIC/ACM WI-IAT*, 2011.

APPENDIX

A. SPARQL ANALYTICAL QUERIES

In this section, we provide a subset of evaluated SPARQL analytical queries with multiple grouping-aggregation constraints. The complete set of evaluated queries and Hive scripts are available on the project website [2].

G5. Retrieve drug-like compounds in PubChem that share common targets with Dexamethasone in the DrugBank (count targets per compound).
```
SELECT ?cid (COUNT(?cid) as ?active_assays {
 ?b CID ?cid; outcome ?a; Score ?s1; gi ?gi .
 ?u gi ?gi; geneSymbol ?g .
 ?di gene ?g; DBID ?dr .
 ?dr Generic_Name "Dexamethasone" .
} GROUP BY ?cid
```

G6. Retrieve compounds in PubChem that are active towards targets in a given pathway (MAPK signalling pathway) in KEGG pathway dataset.
```
SELECT ?cid (COUNT(?cid) as ?active_assays) {
 ?b CID ?cid; outcome ?a; Score ?s1; gi ?gi .
 ?u gi ?gi .
 ?pathway protein ?u; Pathway_name ?pname .
 FILTER regex(?pname,"MAPK signaling pathway","i")
} GROUP BY ?cid
```

G7. Retrieve pathways in the KEGG dataset that contain targets with drugs associated with hepatotoxicity (analyse side-effect hepatomegaly).
```
SELECT ?pid (COUNT(?pid) as ?count) {
 ?sider side_effect ?se; cid ?cid .
 FILTER regex(?se,"hepatomegaly","i")
 ?dr CID ?cid .
 ?target DBID ?dr; SwissProt_ID ?u .
 ?pathway kegg:protein ?u; pathwayid ?pid .
} GROUP BY ?pid
```

MG1. Compare the average price of products per feature vs. price across all features (ProductType1).
```
 SELECT ?f ?sumF ?cntF ?sumT ?cntT {
{ SELECT ?f (COUNT(?pr2) ?cntF) (SUM(?pr2) ?sumF)
 {?p2 type ProductType1; label ?l2; productFeature ?f.
  ?off2 product ?p2; price ?pr2 .
 } GROUP BY ?f
}
{ SELECT (COUNT(?pr) As ?cntT) (SUM(?pr) As ?sumT)
 {?p1 type ProductType1; label ?l1 .
  ?off1 product ?p1; price ?pr .
 } } }
```

MG3. Compare the average price of products per country-feature vs. price per country across all features (for products of type ProductType1).
```
 SELECT ?f ?c ?sumF ?cntF ?sumT ?cntT {
{ SELECT ?f ?c (COUNT(?pr2) ?cntF) (SUM(?pr2) ?sumF)
 {?p2 type ProductType1; label ?l2; productFeature ?f.
  ?off2 product ?p2; price ?pr2; vendor ?v2 .
  ?v2 country ?c .
 } GROUP BY ?f ?c
}
{ SELECT ?c (COUNT(?pr) As ?cntT) (SUM(?pr) As ?sumT)
 {?p1 type ProductType1; label ?l1 .
  ?off1 product ?p1; price ?pr; vendor ?v1 .
  ?v1 country ?c .
 } GROUP BY ?c
} }
```

MG6. Compare the count of targets for a chemical compound and gene combination vs. targets per compound (across all genes).
```
SELECT ?cid ?g1 ?aPerCG ?aPerC {
{ SELECT ?cid ?g1 (COUNT(?cid) as ?aPerCD)
 {?b1 CID ?cid; outcome ?a1; Score ?s1; gi ?gi1 .
  ?u1 gi ?gi1; geneSymbol ?g1 .
  ?di1 gene ?g1; DBID ?dr1 .
 } GROUP BY ?cid ?g1
}
{ SELECT ?cid (COUNT(?cid) as ?aPerG)
 {?b CID ?cid; outcome ?a; Score ?s; gi ?gi .
  ?u gi ?gi; geneSymbol ?g .
  ?di gene ?g; DBID ?dr .
 } GROUP BY ?cid
} }
```

MG9. Compare no. of medline publications per gene vs. total count.
```
SELECT ?gs ?pPerGene ?pT {
{ SELECT ?gs (COUNT(?gs) as ?pPerGene)
 {?g geneSymbol ?gs .
  ?pmid gene ?g; side_effect ?se .
 } GROUP BY ?gs
}
{ SELECT (COUNT(?gs1) as ?pT)
 {?g1 geneSymbol ?gs1 .
  ?pmid1 gene ?g1; side_effect ?se1 .
 } } }
```

MG11. Compare the count of journals funded by grant agencies of a country with the total count of journals published.
```
SELECT ?c ?cntC ?cntT {
{ SELECT ? (COUNT(?g) as ?cntC)
 {?pub journal ?j; grant ?g .
  ?g grant_agency ?ga; grant_country ?c .
 } GROUP BY ?c
}
{ SELECT (COUNT(?g1) as ?cntT)
 {?pub1 journal ?j1; grant ?g1 .
  ?g1 grant_agency ?ga1 .
 } } }
```

MG13. Compare the number of medical subject headings (MeSH) associated per author and publication type with total MeSH per publication type.
```
SELECT ?a ?pty ?perPT ?perAPt {
{ SELECT ?a ?pty (count(?m) as ?perAPT)
 {?pub pub_type ?pty; mesh_heading ?m; author ?a .
  ?a last_name ?ln .
 } GROUP BY ?a ?pty
}
{ SELECT ?pty (count(?m1) as ?perPT)
 {?p1 pub_type ?pty; mesh_heading ?m1; author ?a1 .
  ?a1 last_name ?ln1.
 } GROUP BY ?pty
} }
```

MG16. Compare the number of compounds associated with publications of type "News" (higher selectivity than Journal Articles).
```
SELECT ?ln ?perA ?allA {
{ SELECT ?ln (count(?chem) as ?perA)
 {?pub pub_type "News"; chemical ?ch; author ?a .
  ?a last_name ?ln .
 } GROUP BY ?ln
}
{ SELECT (count(?chem1) as ?allA)
 {?pub1 pub_type "News"; chemical ?ch1; author ?a1 .
  ?a1 last_name ?ln1.
 } } }
```

MG18. Count journal articles per author and grant-awarding country and compare with total journal articles per county (across authors).
```
SELECT ?c ?a ?perC ?perAC {
{ SELECT ?c ?a (count(?g) as ?perAC)
 {?p pub_type "Journal Article"; author ?a; grant ?g.
  ?g grant_agency ?ga; grant_country ?c .
 } GROUP BY ?c ?a
}
{ SELECT ?c (count(?g1) as ?perC)
 {?pub1 pub_type "Journal Article"; grant ?g1 .
  ?g1 grant_agency ?ga1; grant_country ?c .
 } GROUP BY ?c
} }
```