# Virtual: Compressing Data Lake Files

Mihail Stoian
University of Technology Nuremberg
mihail.stoian@utn.de

Alexander van Renen
University of Technology Nuremberg
alexander.van.renen@utn.de

Jan Kobiolka
University of Technology Nuremberg
jan.kobiolka@utn.de

Ping-Lin Kuo
University of Technology Nuremberg
ping-lin.kuo@utn.de

Andreas Zimmerer
University of Technology Nuremberg
andreas.zimmerer@utn.de

Josif Grabocka
University of Technology Nuremberg
josif.grabocka@utn.de

Andreas Kipf
University of Technology Nuremberg
andreas.kipf@utn.de

## ABSTRACT

`Virtual` is an open-source add-on for open storage formats, such as Apache Parquet, that substantially reduces file sizes of relational tables while still ensuring efficient column scans. `Virtual` learns sparse functions within the data and uses them for compression. Our demonstration features an interactive dashboard that allows users to discover these functions in their data and understand how `Virtual` uses them for compression and during query execution.

## 1 INTRODUCTION

With the rise of data lakes, open storage formats such as Apache Parquet [4] and ORC [3] have established themselves as solutions that seem to offer good enough file sizes. For instance, HuggingFace [5], which stores large datasets for training and fine-tuning ML models, has started using Parquet as an alternative storage format. Parquet uses lightweight columnar encoding schemes, such as `FOR`, `DICT`, and `DELTA`, to enable fast scans. Although its decompression is fast, this choice of encoding schemes results in unsatisfactory compression ratios. It is therefore not uncommon to see Parquet combined with a general-purpose compression algorithm such as Snappy [6] or Zstd [7] to further reduce file sizes. However, these general-purpose compression algorithms often have slower decompression times than `FOR` for integers [19] or `FSST` for strings [9]. To this end, recent research [8, 13] has pushed for more efficient formats that aim to preserve high scan throughput while *trying* to offer competitive compression ratios; even so, a general-purpose compression algorithm such as Zstd may still perform better in terms of file size [13].

**"The Hidden Treasure": Functions.** On a philosophical note, this plateau observed in the current file formats and research proposals such as BtrBlocks [13] and FastLanes [8] is due to what we call *data agnosticity*. Specifically, traditional encoding schemes do not take advantage of the inherent dependencies between columns. To understand the situation, let us consider the following example.

*Example 1.1.* Due to a sudden increase in AWS S3 [2] prices, a data engineer is tasked with reducing the storage cost of the company's data stored in Parquet format. The company runs analytical queries to understand country-wide employee satisfaction. For simplicity, consider the following dataset of employee earnings [1]:

| `Regular Earnings` | `Overtime Earnings` | `Total Earnings` |
|---|---|---|
| 32,000 | 300 | 32,300 |
| 60,000 | 0 | 60,000 |
| 10,000 | 500 | 10,500 |

The engineer realizes that the "`Total Earnings`" column is simply the sum of the other two columns and therefore does not need to be stored explicitly. The engineer thus goes ahead and tries to remove this column from all queries and datasets. The column can then be computed on the fly, e.g., by using non-materialized views. However, this necessitates a lot of work, especially as other systems may also depend on it. `Virtual` performs these two steps automatically.

**Problem Complexity.** This "negligence" of the presence of functions in data gave rise to a new research line on *correlation-aware* table compression. The key idea is to exploit these (hidden) functions during compression [10–12, 14–17, 20]. Notably, while our engineer's hand-crafted solution may work for the simple example above, functions within real-world tables are often more complex. `Virtual`, our open-source compression framework, automates both tasks: function discovery and query rewriting.

**System.** `Virtual` [20] offers a novel approach to further reducing Parquet file sizes by leveraging dependencies between columns while preserving efficient column scans. In fact, it acts as an add-on for *any* open storage format. Behind the scenes, `Virtual` searches for sparse functions, i.e., functions that depend on as few columns as possible, which ensures efficient column scans. As of now, `Virtual` automatically finds sparse *linear* functions. Beyond the theoretical guarantees [20], dealing with real data always incurs several other issues: missing values, change of formats within the same column, type-preserving expressions, etc. Supporting all these aspects necessitates a careful engineering effort. This paper showcases `Virtual` as a ready-to-use add-on for Apache Parquet [4], along with an interactive dashboard that helps the user understand how the system *virtualizes* the table and rewrites the queries.

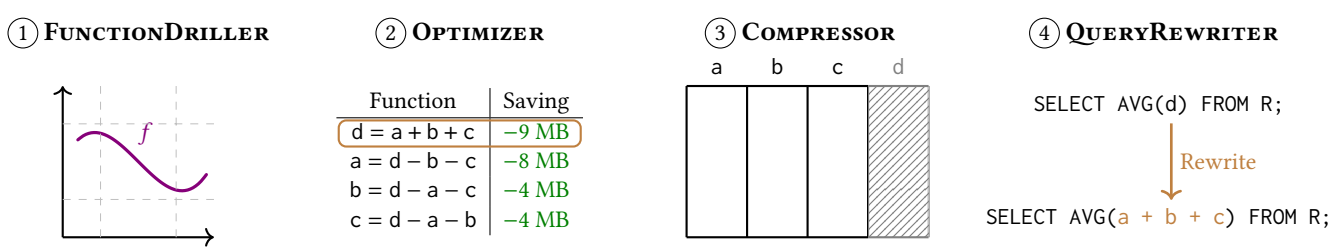


**Figure 1: The four main components of `Virtual`: The FunctionDriller discovers the underlying functions in the table. The Optimizer selects a subset of these to minimize the file size and the Compressor virtualizes the corresponding column(s). Finally, the QueryRewriter ensures that queries are translated into the equivalent form w.r.t. the functions.**

**Contribution.** In summary, we demonstrate `Virtual`'s

(1) dashboard where users can upload their datasets to virtualize them and write analytical SQL queries as usual,
(2) interactive game that allows users to find the functions themselves and better understand the complexity of the underlying problem,
(3) easy-to-use API that users can adopt in their data pipelines.

## 2 SYSTEM OVERVIEW

`Virtual` is composed of four independent components:

(1) FunctionDriller,
(2) Optimizer,
(3) Compressor,
(4) QueryRewriter,

visualized in Fig. 1. Let us see their intertwining in `Virtual`.

**Workflow.** The FunctionDriller is responsible for finding the hidden functions in the dataset. In `Virtual`, we discover *sparse* functions, i.e., functions that do not depend on too many reference columns [20]. Currently, we support automatic detection of sparse *linear* functions and a novel $k$-regression that aims to fit the data with multiple linear regression lines.

Next, the Optimizer estimates the compression benefits of each function using a sample of the data and selects a subset of the discovered functions that minimizes the storage space. To understand why this is necessary, consider our initial Example 1.1. There are three functions present in the data:

`Total Earnings` = `Regular Earnings` + `Overtime Earnings`
`Regular Earnings` = `Total Earnings` − `Overtime Earnings`
`Overtime Earnings` = `Total Earnings` − `Regular Earnings`

First, note that only a single function from this set can be used; otherwise, we risk creating a cyclic dependency between functions. Second, depending on the actual values in the columns, they might come with different compression gains. The Optimizer tries to select the optimal subset of non-conflicting functions; this takes exponential time in the number of functions. If this number is too large, the optimizer resorts to a greedy strategy.

The Compressor takes the chosen subset of functions and compresses the table. At a high level, it applies the identified functions, attempts to resolve any inconsistencies with the original values, and drops the columns that are computed by these functions. This is probably the most engineering-heavy part of our system since we have to deal with precision issues, the presence of `NULL`s, etc. In particular, note that we aim for a *lossless* reconstruction, which is guaranteed by several auxiliary columns:

(a) `offset` accounts for the possible errors in the function (we minimize via the $\mathcal{L}_2$-error),
(b) `is_null` marks whether the original value was `NULL`,
(c) `outlier` stores the original value in case a reference column had a `NULL`-value on the respective row,
(d) and `switch` stores the index of the regression line to use; only used if $k$-regression is used.

Note that the auxiliary columns themselves are guaranteed to be either small (`is_null`, and `offset` due to $\mathcal{L}_2$-minimization) or sparse (`outlier`), and can thus be compressed well. Next, we show how these components are mirrored in `Virtual`'s interactive dashboard.

### 2.1 Dashboard

The dashboard captures and exhibits `Virtual`'s main functionality. In Fig. 2, we show the dashboard when virtualizing the full dataset from Example 1.1 [1]; the header of the dashboard offers two virtualizable tables, one of them being TPC-H `lineitem`.

For ease of explanation, the dashboard is split into four areas (A)-(D) and partitioned along two parts: table preview and statistics (A)-(B), and an interactive SQL query interface (C)-(D).

**Preview and Statistics.** The user can preview the table (area (A)) and even how `Virtual` stores it internally via a toggle switch "Virtual view". On the right, one can analyze the functions drilled from the table and how `Virtual` compares to CSV and Parquet in terms of file size (area (B)).

**Query Interface.** We expose `Virtual`'s query support via a verbose query interface (area (C)). The user can ask arbitrary SQL queries in a console. Alternatively, a click on one of the functions from area B – in our case "`Total Earnings = Regular Earnings + Overtime Earnings`" – generates a sample SQL query on the corresponding virtualized column (`Total Earnings`). To understand how the QueryRewriter modifies the query, one can toggle "Show rewrite", which splits the original console into two and shows on the right the query that will be actually run by `Virtual`. In our example, the rewriter replaced column `Total Earnings` by its functional representation. To understand the slowdown over Parquet[1] – since we now read two columns instead of a single one – one can also run the query from a vanilla Parquet file, i.e., on the unvirtualized table. The query time, i.e., the time taken to perform the rewrite and execute the rewritten query, and the query result are shown on the right (area (D)).

In the next section, we introduce `Virtual`'s API, which powers our dashboard.

[1]We use DuckDB [18] as the execution engine.

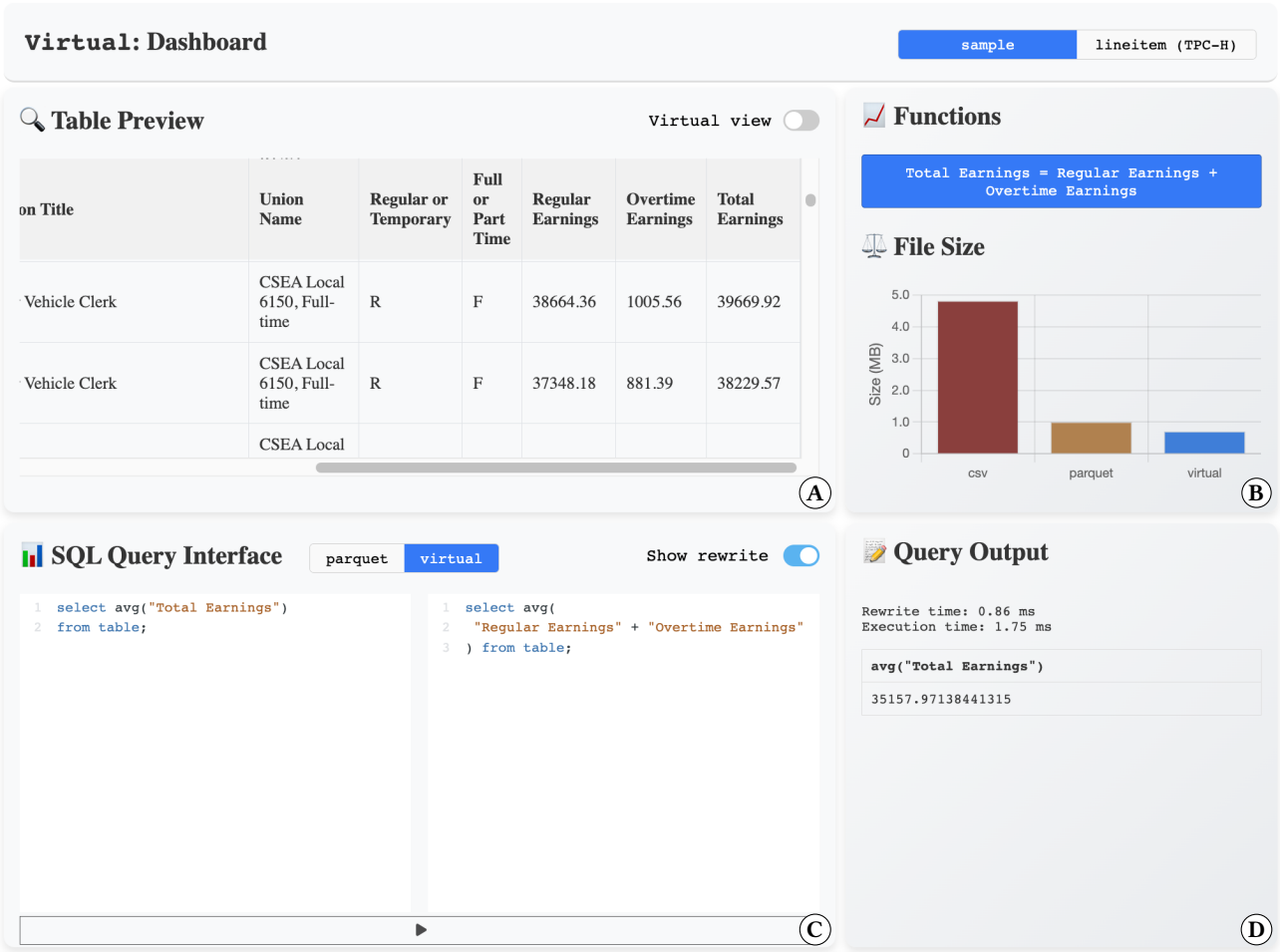


**Figure 2: Dashboard: Users can upload their datasets and see how `Virtual` uses functions for column compression and during query execution.**

## 2.2 API

`Virtual` is designed with an easy-to-use API so that it can act as an add-on for any open storage format. We exemplify it for Parquet, due to its first-class support in DuckDB [18] and Pandas:

- `to_format`: Takes as input a table (`pd.DataFrame`, CSV or any supported file format), virtualizes it, and saves it to a file of the specified file format.
- `query`: Rewrites and executes the SQL query on the file.

Let us exemplify them, as used in our dashboard (Sec. 2.1).

**Instant Virtualization.** In this use-case, the user reads a dataframe, operates on it, and finally stores it as Parquet via `Virtual`.

```
import pandas as pd
import virtual

% Read data.
df = pd.read_csv('file.csv')

% User operations.
df = ...

% Virtualize + save to Parquet.
virtual.to_format(df, 'file_virtual.parquet')
```

**Query.** The virtualized Parquet file can be queried via SQL, using the given execution engine (in this example, DuckDB [18]). This function performs the necessary rewrites while preserving the semantic of the original query, i.e., the output yields the data types and column namings from the original query.

```
import virtual

% Query the file.
virtual.query(
  '''select avg(price)
  from read_parquet("file_virtual.parquet")
  where year >= 2024''',
  engine = 'duckdb'
)
```

**Other Features.** Optionally, users can drill the functions on their own and export them to a JSON file via `virtual.train`. We also provide a handy `from_format` method that converts the virtualized parquet file back into a `pd.DataFrame`.

## 3 DEMONSTRATION

Our live demo is structured in two phases. The first part consists of a game featuring a leaderboard where session participants are invited to interactively explore the main challenge behind

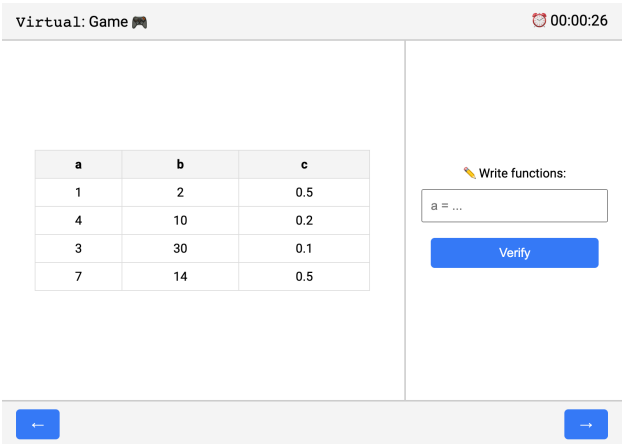


**Figure 3: Game: The demo session participant has to discover the functions hidden in the data. The results are displayed in a leaderboard.**

`Virtual`: finding functions in the data. In the second phase, visitors can interact with the main dashboard and virtualize datasets.

**Game with Leaderboard.** First, session participants are invited to play an interactive game to grasp the complexity of the problem of finding suitable functions. We will show five small tables of varying difficulty, and the participant has to input the function(s) that could be hidden in the table. A leaderboard will rank the participants by the number of points accumulated and the time taken to complete the game. This will be the first webpage that the participants will interact with. A screenshot of this game for a sample table is shown in Fig. 3.

**Dashboard.** The main entry into the dashboard is a drag-and-drop / choose-file page where the participant can upload their favorite CSV file and have it virtualized; we even provide a way to enter a custom `pd.DataFrame`. If no specific file is wished for, we will have prepared two datasets of different difficulty, to also show the table layout in the case where `Virtual` finds a $k$-regressor [20]. We will let them write arbitrary queries and interact with the query rewriter.

## REFERENCES


[1] 2020. Broome County Annual Employee Earnings: Beginning 2009. https://catalog.data.gov/dataset/broome-county-annual-employee-earnings-beginning-2009.

[2] 2024. Amazon Simple Storage Service (Amazon S3). https://aws.amazon.com/s3/

[3] 2024. Apache ORC. Apache Software Foundation. https://orc.apache.org/

[4] 2024. Apache Parquet. Apache Software Foundation. https://parquet.apache.org/

[5] 2024. Hugging Face. https://huggingface.co/

[6] 2024. Snappy. https://github.com/google/snappy.

[7] 2024. Zstandard. https://github.com/facebook/zstd.

[8] Azim Afroozeh and Peter A. Boncz. 2023. The FastLanes Compression Layout: Decoding >100 Billion Integers per Second with Scalar Code. *Proc. VLDB Endow.* 16, 9 (2023), 2132–2144. https://doi.org/10.14778/3598581.3598587

[9] Peter A. Boncz, Thomas Neumann, and Viktor Leis. 2020. FSST: Fast Random Access String Compression. *Proc. VLDB Endow.* 13, 11 (2020), 2649–2661. http://www.vldb.org/pvldb/vol13/p2649-boncz.pdf

[10] Bogdan Ghita, Diego G. Tomé, and Peter A. Boncz. 2020. White-box Compression: Learning and Exploiting Compact Table Representations. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p4-ghita-cidr20.pdf

[11] Thomas Glas. 2023. Exploiting Column Correlations for Compression. https://homepages.cwi.nl/~boncz/.

[12] Amir Ilkhechi, Andrew Crotty, Alex Galakatos, Yicong Mao, Grace Fan, Xiran Shi, and Ugur Çetintemel. 2020. DeepSqueeze: Deep Semantic Compression for Tabular Data. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 1733–1746. https://doi.org/10.1145/3318464.3389734

[13] Maximilian Kuschewski, David Sauerwein, Adnan Alhomssi, and Viktor Leis. 2023. BtrBlocks: Efficient Columnar Compression for Data Lakes. *Proc. ACM Manag. Data* 1, 2 (2023), 118:1–118:26. https://doi.org/10.1145/3589263

[14] Hao Liu, Yudian Ji, Jiang Xiao, Haoyu Tan, Qiong Luo, and Lionel M. Ni. 2017. TICC: Transparent Inter-Column Compression for Column-Oriented Database Systems. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, CIKM 2017, Singapore, November 06 - 10, 2017*, Ee-Peng Lim, Marianne Winslett, Mark Sanderson, Ada Wai-Chee Fu, Jimeng Sun, J. Shane Culpepper, Eric Lo, Joyce C. Ho, Debora Donato, Rakesh Agrawal, Yu Zheng, Carlos Castillo, Aixin Sun, Vincent S. Tseng, and Chenliang Li (Eds.). ACM, 2171–2174. https://doi.org/10.1145/3132847.3133077

[15] Hanwen Liu, Mihail Stoian, Alexander van Renen, and Andreas Kipf. 2024. Corra: Correlation-Aware Column Compression. In *Proceedings of Workshops at the 50th International Conference on Very Large Data Bases, VLDB 2024, Guangzhou, China, August 26-30, 2024*. VLDB.org. https://vldb.org/workshops/2024/proceedings/CloudDB/clouddb-2.pdf

[16] Hao Liu, Jiang Xiao, Xianjun Guo, Haoyu Tan, Qiong Luo, and Lionel M. Ni. 2017. Cuttle: Enabling Cross-Column Compression in Distributed Column Stores. In *Web and Big Data - First International Joint Conference, APWeb-WAIM 2017, Beijing, China, July 7-9, 2017, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 10367)*, Lei Chen, Christian S. Jensen, Cyrus Shahabi, Xiaochun Yang, and Xiang Lian (Eds.). Springer, 219–226. https://doi.org/10.1007/978-3-319-63564-4_18

[17] Xi Lyu, Andreas Kipf, Pascal Pfeil, Dominik Horn, Jana Giceva, and Tim Kraska. 2023. CorBit: Leveraging Correlations for Compressing Bitmap Indexes. In *VLDB Workshops (CEUR Workshop Proceedings, Vol. 3462)*. CEUR-WS.org.

[18] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 1981–1984. https://doi.org/10.1145/3299869.3320212

[19] Julia Spindler, Philipp Fent, Adrian Riedl, and Thomas Neumann. [n. d.]. Can Delta Compete with Frame-of-Reference for Lightweight Integer Compression? *Proceedings of the VLDB Endowment. ISSN* 2150 ([n. d.]), 8097.

[20] Mihail Stoian, Alexander van Renen, Jan Kobiolka, Ping-Lin Kuo, Josif Grabocka, and Andreas Kipf. 2024. Lightweight Correlation-Aware Table Compression. In *NeurIPS 2024 Third Table Representation Learning Workshop*. https://openreview.net/forum?id=z7eIn3aShi