Lightweight Correlation-Aware Table Compression

Mihail Stoian, Alexander van Renen, Jan Kobiolka, Ping-Lin Kuo,
Josif Grabocka, Andreas Kipf
{mihail.stoian, alexander.van.renen, jan.kobiolka, ping-lin.kuo,
josif.grabocka, andreas.kipf}@utn.de
University of Technology Nuremberg

Abstract

The growing adoption of data lakes for managing relational data necessitates efficient, open storage formats that provide high scan performance and competitive compression ratios. While existing formats achieve fast scans through lightweight encoding techniques, they have reached a plateau in terms of minimizing storage footprint. Recently, correlation-aware compression schemes have been shown to reduce file sizes further. Yet, current approaches either incur significant scan overheads or require manual specification of correlations, limiting their practicability. We present `Virtual`, a framework that integrates seamlessly with existing open formats to automatically leverage data correlations, achieving substantial compression gains while having minimal scan performance overhead. Experiments on `data.gov` datasets show that `Virtual` reduces file sizes by up to 40% compared to Apache Parquet.

1 Introduction

When a comet passes the Earth, mankind carefully prepares well in advance; months, if not years. We see something similar with the constant shift to the cloud: Large relational tables are moving to the cloud, and we are not there yet in terms of efficient processing times. This is partly because current storage formats, such as Apache Parquet, cannot keep up with the ever-increasing data sizes. Simply put, scan performance and compression ratios are still poor. In fact, Parquet itself applies general-purpose compression, e.g., Snappy, on top to improve its storage footprint further. As a result, latest research, e.g., BtrBlocks [1], FastLanes [2], tries to adapt to the new setting by ensuring two main requirements: (`i`) column scans are fast, (`ii`) compression ratios remain competitive.

While requirement (`i`) has been pushed to its limits, the second still awaits its solution. The key insight is that existing techniques use only simple encoding schemes [3-6] which naturally leads to a plateau in terms of minimizing storage footprint.

State of the Art. For this very reason, recent research proposes a change in mindset: Instead of relying only on simple encoding schemes such as `RLE` or `DELTA`, leverage correlations between columns that are present in real-world datasets [7-11] and represent them in a compact way; the standard encoding schemes can be applied on top. This is known as semantic or correlation-aware compression and has been shown to enable (`ii`). However, existing solutions for numeric columns either do not satisfy (`i`) or are not user-friendly: DeepSqueeze [7] is based on an auto-encoder architecture that naturally makes a column dependent on all the other columns; this incurs a slowdown during column scans and violates (`i`). Corra [10], while lightweight in itself, relies on user-defined functions to actually be able to compress. C3 [11] automates this process, but only applies a simple linear regression to column-pairs for its numeric columns, leaving compression potential unexplored.

Contribution. We introduce `Virtual`, a lightweight framework that *automatically* detects *sparse* functions inherently present in a given table, enabling both (`i`) fast scan times and (`ii`) competitive

3rd Table Representation Learning Workshop (TRL 2024)

compression ratios. `Virtual` is designed to be a plug-in for open storage formats. In particular, we show saving factors of up to 40% over Parquet on `data.gov` tables and a minimal slowdown of column scans on the virtualized columns. `Virtual` is open source and available at `https://github.com/utndatasystems/virtual`.

2 Lightweight Correlation-Aware Compression

`Virtual` has three main components: (a) FUNCTIONDRILLER (Sec. 2.1), (b) OPTIMIZER (Sec. 2.2), and (c) COMPRESSOR (Sec. 2.2). This corresponds to three natural steps: (a) Find the inherent functions in the table, (b) choose which functions to select from the previously found set, and then (c) perform the actual compression using the functions. In the following, we describe (a) in more detail.

2.1 FUNCTIONDRILLER

Our function "driller" determines which tuples of columns are correlated and the associated functions.

(Simple) Example. As an introductory example, consider the real-world dataset "*Monthly Modal Time Series*" within `data.gov`: It has a total of thirteen column names prefixed with "*Total*". Not surprisingly, all these columns are the sum of specific columns in the dataset – we refer to these as *reference* columns. The to-be-virtualized columns are referred to as *target* columns. Yet, no current open storage format exploits this correlation. As we show in this work, considering these column correlations results in high saving rates.

(Complex) Example. Not all correlations are as explicit as the previous one. Prior work has shown that the well-studied Taxi [12] table has multiple functions underlying its `total_amount` column [10]. This is the motivation behind what will be introduced next as K-regression. Note that Liu et al. [10] identified the different functions by hand, while `Virtual` finds them automatically.

Drilling via K-Regression. To find the above functions in a principled manner, we fix a target column $y \subset T$ from a given table T and find an estimator using the remaining columns, $X := T \setminus y$. Specifically, we use a regression model known in the literature as *cluster-wise* linear regression [13, 14]. The task is to partition the table records into K clusters such that the sum of squared errors of the linear regressions across clusters is minimized. Let $(C_1, C_2, \ldots, C_K)$ represent the cluster partitions, where C_k represents the set of row indices assigned to the k-th cluster. Furthermore, let $\boldsymbol{w}_k, \beta_k$ be the weight vector and the intercept for the linear model of cluster C_k.

We initialize the model by randomly assigning each data row i to one of K distinct clusters and subsequently estimate a linear regression model for each cluster. In the **first step**, we assign each row i to the optimal cluster $k^{(i)}$ computed as the cluster whose linear model best approximates the target column y_i of the table row i:

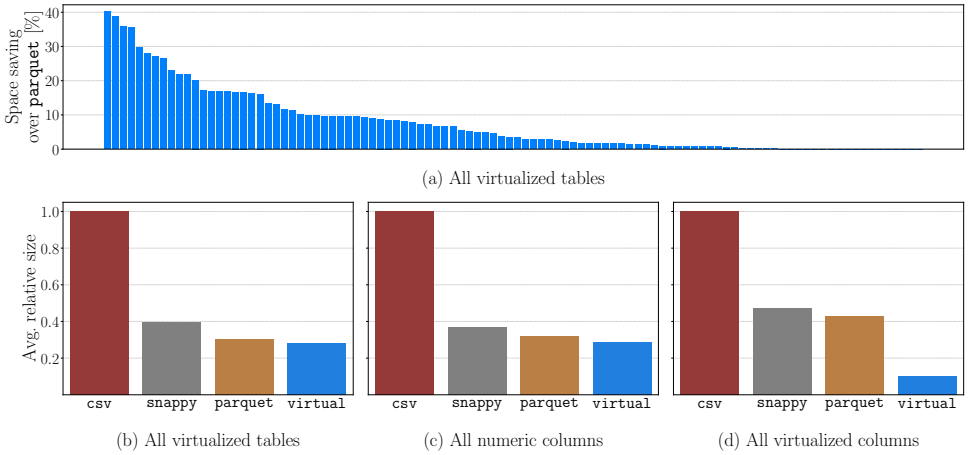
$$\forall i: \quad C_{k^{(i)}} \leftarrow C_{k^{(i)}} \cup \{i\}, \text{ where } k^{(i)} = \underset{k \in \{1,\ldots,K\}}{\arg\min} \left(y_i - X_i^\top \boldsymbol{w}_k - \beta_k\right)^2. \tag{1}$$

Given the cluster assignments, in the **second step**, we then optimize the linear regression models as:

$$\underset{\boldsymbol{w},\beta}{\arg\min} \sum_{k=1}^{K} \sum_{i \in C_k} \left(y_i - X_i^\top \boldsymbol{w}_k - \beta_k\right)^2. \tag{2}$$

The method iterates the two steps of Eq. (1) and Eq. (2) until convergence (for practical reasons, only on a randomly selected sample of 10^4 rows). However, using this naively does not satisfy one of our main requirements: "*scans should be fast*". To see why this is the case, note that a plain linear regressor is not optimized for a small number of reference columns. We alleviate this issue with a sparsity regularization approach.

Why Sparseness Matters. Correlation-aware compression, by design, introduces overhead for subsequent table scans, which may lead to impractical downstream data analyses. To understand this limitation, consider the following SQL query (a), which computes the average of the total amounts across all taxi rides [12]:



(a) All virtualized tables

(b) All virtualized tables (c) All numeric columns (d) All virtualized columns

Figure 1: Improvement over Parquet+Snappy (`parquet`) across 103 virtualized `data.gov` tables

(a) `select avg(total_amount) from taxi;` 2.7 ms

(b) `select avg(fare_amount + mta_tax + tip_amount) from taxi;` ↑6.5 ms

Naturally, if `total_amount` is now calculated by the sum of other columns as in SQL query (b), the execution engine[1] needs to load these columns to evaluate the expression. As the example shows, this leads to extra I/O which slows down the query by a linear factor depending on the number of reference columns. Hence, to minimize scan overhead, we should aim for the *minimum* number of reference columns that have to be loaded.

Formally, this leads to a *sparse* linear regression [16], i.e., the number of non-zero entries of the estimator should be minimized using a L_0 penalty. Hence, instead of minimizing Eq. (2), we have:

$$\underset{\boldsymbol{w},\beta}{\arg\min} \sum_{k=1}^{K} \sum_{i \in C_k} \left(y_i - X_i^\top \boldsymbol{w}_k - \beta_k\right)^2 + \lambda \|\boldsymbol{w}\|_0, \tag{3}$$

where $\|\boldsymbol{w}\|_0$ is defined as the number of non-zero coefficients of the coefficient vector $\boldsymbol{w}$; notably, there is constraint for the intercept β since it does not involve the loading of any reference column.

2.2 Optimizer & Compresser

Conflicting Functions. Once the K-regressor has been found, we can start virtualizing the columns. Yet, we are still missing an important aspect: Not all columns can actually be virtualized. To see why this is the case, consider the following example: Column "`a`" can be computed from "`b` + `c`". However, this also holds for "`b`" and "`c`", e.g., "`b` = `a` − `c`". Indeed, virtualizing any column-pair would lead to a cyclic dependency between the columns. We use an optimizer that estimates the space saving of the found functions and greedily chooses a valid subset to maximize the estimated gain.

Compression. The selected target columns output by the optimizer can now be virtualized. The auxiliary columns to store are as follows: (a) `offset` — which stores the offset to the *closest* of the K regressors, and (b) `switch` — which indicates in $\log_2 K$ bits which of the K regressors has been chosen.[2] However, when applying correlation-aware compression to real-world tables, the missing value issue becomes more stringent. Intuitively, they add another layer of complexity, as evaluating expressions on `NaN` values results in such a value. Unlike prior work, we choose not to drop these rows but instead also store the following auxiliary columns: (c) `outlier` — in case one of the reference values is missing (`NaN`), and (d) `is_nan` — which indicates whether the original value was itself a `NaN`. This layout ensures that data can be reconstructed in a lossless manner.

[1] In this case, we used DuckDB [15] and ingested the first 10M rows of the table.

[2] In the case of $K = 1$, this auxiliary column is not considered.

3 Experiments

We perform a large-scale evaluation to understand the applicability of `Virtual` to real-world tables. We run on a `c5d.4xlarge` instance with 16 vCPUs, 32 GB of memory, and 500 GB of disk space. We downloaded the most popular CSV tables from `data.gov` [17], obtaining 1,226 datasets.[3] As relational tables are usually stored in data blocks, we restrict ourselves to the first 10^6 rows of each table. Moreover, note that we guarantee a *lossless* compression, as is common in databases.

Compression Ratios. Of the 1,226 tables, we were able to find correlations in 157 of them (we selected regressors with error less than 1). We also dropped tables with less than 1,000 rows, since, in these scenarios, the Parquet files were larger than the original CSV files. We are left with 103 tables, whose space savings over Parquet+Snappy we show sorted in Fig. 1(a). The largest space saving of 40% is for a geo-location dataset [18]. In Fig. 1(b-d), we show relative sizes to CSV, with Fig. 1(d) best expressing the effect of virtualization on a total of 328 columns.

Scan Times. We enable fast column scans for correlation-aware compression by relying on sparse K-regressors. In Fig. 2, we show the slowdown over Parquet+Snappy when summing up the values of virtualized columns directly from the files generated by default Parquet and `Virtual`, respectively.[4] Since the auxiliary columns (Sec. 2.2) are lightweight (the `offset` is minimized via Eq. (2)), the overhead scales linearly with the number of reference columns.

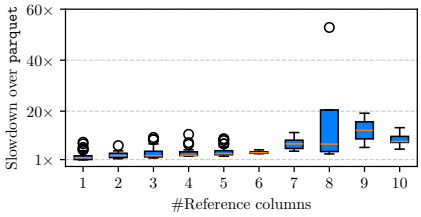


Figure 2: Column scan slowdowns

Comparison to `DeepSqueeze` [7]. We also compare to `DeepSqueeze`, an auto-encoder architecture that provides excellent compression ratios. We use its publicly available re-implementation [19] (note that this only supports numeric columns; Ilkhechi et al. [7] do not specify how many distinct values make up a categorical column). Since `DeepSqueeze` is lossy by design, we made the following changes: We (1) removed input quantization – essentially the "lossy" component in `DeepSqueeze`, (2) added offsets post-rescaling, (3) rounded the results to the correct precision, and (4) stored rows with `NaN`s in a separate Parquet file. Regarding (2), we found that adding offsets after rescaling empirically reduced the file size for integer columns (compared to storing small floating-point offsets). In Tab. 1, we show the file size and column scan latency for three tables from `data.gov`. In particular, `DeepSqueeze` becomes impractical in terms of column scans as it has to load the *entire* latent space to perform a single column scan.

Metric	File size [MB]		Column scan [ms]	
Approach	`DeepSqueeze`	`Virtual`	`DeepSqueeze`	`Virtual`
`DASH` [20]	26.92	9.18 ↓	1484.66	18.43 ↓
`Taxi trips` [12]	19.68	15.65 ↓	515.62	74.40 ↓
`Weekly Counts[...]` [21]	3.50	1.20 ↓	663.37	8.95 ↓

Table 1: Comparison to `DeepSqueeze` [7] on table size and virtualized column scan latency

4 Discussion & Future Work

`Virtual` provides high space savings over Apache Parquet while having a negligible scan overhead. We believe that *arbitrary* sparse functions, such as "tiny" neural networks, which ensure that the number of reference columns is minimized, can improve the current compression factors further. Moreover, scan performance could be accelerated by introducing additional metadata, such as min-max bounds, for the virtualized columns in the storage layer itself.

[3] A number of 150 datasets had downloading issues, showing a pending update; we already excluded these.
[4] Note that for robust measurements, we duplicate the rows until each table reaches 1M rows.

References

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

[2] Azim Afroozeh and Peter A. Boncz. The FastLanes Compression Layout: Decoding >100 Billion Integers per Second with Scalar Code. *Proc. VLDB Endow.*, 16(9):2132–2144, 2023. doi: 10.14778/3598581.3598587. URL https://www.vldb.org/pvldb/vol16/p2132-afroozeh.pdf.

[3] Daniel Lemire and Leonid Boytsov. Decoding billions of integers per second through vectorization. *Softw. Pract. Exp.*, 45(1):1–29, 2015. doi: 10.1002/SPE.2203. URL https://doi.org/10.1002/spe.2203.

[4] Patrick Damme, Dirk Habich, Juliana Hildebrandt, and Wolfgang Lehner. Lightweight Data Compression Algorithms: An Experimental Survey (Experiments and Analyses). In Volker Markl, Salvatore Orlando, Bernhard Mitschang, Periklis Andritsos, Kai-Uwe Sattler, and Sebastian Breß, editors, *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017*, pages 72–83. OpenProceedings.org, 2017. doi: 10.5441/002/EDBT.2017.08. URL https://doi.org/10.5441/002/edbt.2017.08.

[5] Patrick Damme, Annett Ungethüm, Juliana Hildebrandt, Dirk Habich, and Wolfgang Lehner. From a comprehensive experimental survey to a cost-based selection strategy for lightweight integer compression algorithms. *ACM Trans. Database Syst.*, 44(3):9:1–9:46, 2019. doi: 10.1145/3323991. URL https://doi.org/10.1145/3323991.

[6] Panagiotis Liakos, Katia Papakonstantinopoulou, and Yannis Kotidis. Chimp: Efficient lossless floating point compression for time series databases. *Proc. VLDB Endow.*, 15(11):3058–3070, 2022. doi: 10.14778/3551793.3551852. URL https://www.vldb.org/pvldb/vol15/p3058-liakos.pdf.

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

[8] Bogdan Ghita, Diego G. Tomé, and Peter A. Boncz. 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, 2020. URL http://cidrdb.org/cidr2020/papers/p4-ghita-cidr20.pdf.

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

[10] Hanwen Liu, Mihail Stoian, Alexander van Renen, and Andreas Kipf. Corra: Correlation-Aware Column Compression. *CoRR*, abs/2403.17229, 2024. doi: 10.48550/ARXIV.2403.17229. URL https://doi.org/10.48550/arXiv.2403.17229.

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

[12] 2018 Yellow Taxi Trip Data, 2023. URL https://catalog.data.gov/dataset/2018-yellow-taxi-trip-data.

[13] Wayne S DeSarbo and William L Cron. A Maximum Likelihood Methodology for Clusterwise Linear Regression. *Journal of classification*, 5:249–282, 1988.

[14] H. Späth. Algorithm 39 Clusterwise Linear Regression. *Computing*, 22(4):367–373, Dec 1979. ISSN 1436-5057. doi: 10.1007/BF02265317. URL https://doi.org/10.1007/BF02265317.

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

[16] Hussein Hazimeh, Rahul Mazumder, and Ali Saab. Sparse Regression at Scale: Branch-and-Bound Rooted in First-Order Optimization. *Math. Program.*, 196(1):347–388, 2022. doi: 10.1007/S10107-021-01712-4. URL https://doi.org/10.1007/s10107-021-01712-4.

[17] U.S. Government. Data.gov, 2024. URL https://data.gov. Download date: 2024-07-11.

[18] BLM ES PLSSPoint, 2024. URL https://catalog.data.gov/dataset/blm-es-plsspoint-11ef7.

[19] Mike Xydas. Deepsqueeze, 2021. URL https://github.com/MikeXydas/DeepSqueeze.

[20] DASH - Global School-based Student Health Survey (GSHS), 2023. URL https://catalog.data.gov/dataset/dash-global-school-based-student-health-survey-gshs.

[21] Weekly Counts of Death by Jurisdiction and Select Causes of Death, 2023. URL https://data.cdc.gov/NCHS/Weekly-Counts-of-Death-by-Jurisdiction-and-Select-/u6jv-9ijr/about_data.