Short Paper

# Workload-Driven and Robust Selection of Compression Schemes for Column Stores

Martin Boissier
Hasso Plattner Institute
Potsdam, Germany
martin.boissier@hpi.de

Max Jendruk
Hasso Plattner Institute
Potsdam, Germany
max.jendruk@hpi-alumni.de

## ABSTRACT

Modern main memory-optimized column stores employ a variety of compression techniques. Deciding for one compression technique over others for a given memory budget can be challenging since each technique has different trade-offs whose impact on large workloads is not obvious. We present an automated selection framework for compression configurations. Most database systems provide means to automatically choose a compression configuration but lack two crucial properties: The compression selection cannot be constrained (e.g., by a given storage budget) and robustness of the compression configuration is not considered. Our approach uses workload information to determine robust configurations under the given constraints. The runtime performance of the various compression techniques is estimated using adapted regression models.

## 1 COLUMN COMPRESSION IN HYRISE

Two of the main driving forces of current database development – both industrial and research – are autonomous database systems and cloud-based installations. Both topics are strongly connected as database vendors are increasingly interested in optimizing their operational costs for large self-hosted database installations.

One way to lower the costs – especially for main memory-optimized database systems – is to reduce the memory consumption of large databases. Such a reduction allows storing databases on smaller and thus less expensive server machines or adding more instances to a shared server. But the sheer size of large cloud installations hampers manual optimization of compression configurations by database administrators. This development has recently sparked the research on autonomous database systems.

The work presented in this paper is an intermediate step to approach the issue of optimizing memory consumption while still retaining the performance advantages of main memory-optimized databases. When cost considerations are gaining importance, the optimization objective for compression configurations is less runtime performance rather than to retain the current runtime performance while minimizing the storage requirements. With the goal of automatically finding a compression configuration for a given memory budget, this project intends to provide the building blocks for autonomous systems.

The area of data compression has been thoroughly studied for decades in database research. Virtually all modern database systems implement various techniques to compress data and most commercial systems further provide means to adjust the compression level (e.g., Oracle's declarative policies for the *automatic data compression* (ADO), cf. [12], or SQLServer's *database engine tuning advisor* (DTA), cf. [11]). However, we see two distinct issues that remain open from a research perspective: **(i)** workload- and constraint-based compression configurations and **(ii)** determination of configurations whose runtime performance is robust to changing workloads.

We present and discuss the three main components in our research database Hyrise [10] with which we approach workload-driven and robust compression configurations:

- We introduce Hyrise's compression framework which implements an efficient and maintainable interface for various column compression techniques (Section 2).
- We present our runtime estimation, which predicts the performance of compression techniques (Section 3).
- We discuss the applicability of existing approaches for the optimization of physical database designs and how they perform for the task of compression selection (Section 4).

## 2 COLUMN COMPRESSION FRAMEWORK

Virtually every database management system for hybrid transactional and analytical processing (HTAP) employs a variety of compression schemes. Besides the advantage of reducing the main memory footprint, light-weight compression can even improve runtime performance, e.g., by reducing the memory traffic (cf. [2, 4]) or broadening applicability of vectorization (cf. [17]).

But supporting a variety of compression schemes is challenging as it needs to balance maintainability and efficiency. Most existing approaches optimize either (i) for performance while hampering maintainability and increasing complexity or (ii) provide unified interfaces for improved maintainability which potentially introduces runtimes issues.

### 2.1 Hyrise's Storage Concept

Hyrise is a main memory-optimized database with a column-major storage format [10]. Each table in Hyrise is horizontally partitioned into $n$ chunks with a predefined maximum size. Each attribute of a table is hence distributed over all chunks whereby a column in a chunk is referred to as a segment. Modifications (i.e., insertions or MVCC-enabled updates) are appended to the most recent mutable chunk. When this chunk reaches its size limit, the chunk is considered immutable and a new mutable chunk is created. Immutable chunks might be compressed asynchronously. Hyrise encodes and compresses segments independently.

### 2.2 Balancing Performance and Maintainability

There are multiple approaches to integrate column compression schemes into the database system. One is to decompress vectors before an operator accesses the data, eliminating the need to handle different compression schemes in every operator. While this approach might be sufficient for analytical purposes which are dominated by sequential operations, it is not feasible for HTAP processing where single row accesses are frequent.

Another approach is to adapt operators so that they can directly execute on compressed data using late materialization. This approach lowers the memory bandwidth required and further allows to exploit encoding-specific optimizations (e.g., early exiting a predicate when the searched value does not exist in the dictionary). While this approach promises the best performance, it also increases maintenance efforts significantly.

The middle ground between the upfront decompression of segments and encoding-specialized operators are abstraction layers that provide a unified interface to access data (cf. [2, 14]). However, this approach usually introduces dynamic polymorphism and thus virtual method calls per tuple, which are prohibitively expensive in analytical scenarios [5]. Dynamic polymorphism adds instructions, impedes cache utilization, and further hinders compilers to automatically apply vectorization primitives.

## 2.3 Integrating Compression Schemes

To overcome the issues mentioned in the previous section, Hyrise implements an efficient abstraction layer that provides a unified interface while still allowing encoding-specific optimization when desirable. The implementation uses *zero-cost abstractions* based on C++ metaprogramming and templating.

The column compression framework handles both encoding as well as decoding of compressed segments. The framework separates the various concerns by splitting data storage, encoding, and decoding into separate components. To decode columns, Hyrise uses C++'s iterator concept which – amongst other advantages – allows to apply algorithms of the standard library as if data would simply reside unencoded in an `std::vector`. Moreover, iterators can have state which enables block-based compression schemes to cache the most recently decoded block for potential upcoming accesses to the same block.

In Hyrise, column segments are usually accessed in two ways: fully sequentially or semi-randomly via a position list. As a consequence, each compression scheme provides these two access paths via a *sequential iterator* and a *point-access iterator* which accepts a position list. The impact of providing a positional access path over upfront decompression is shown in Figure 1.

Following the separation presented in [8], Hyrise distinguishes between logical-level and physical-level compression techniques and allows to cascade them. The logical-level techniques currently implemented are dictionary, frame of reference, and run length encoding. The physical-level techniques include fixed-size byte-aligned (FSBA) compression and SIMD-BP128 compression (cf. [8] for more details on the mentioned techniques).

*Implementation Aspects:* Hyrise is written using C++17 and uses of Boost Hana[1] for metaprogramming. To provide static interfaces within the encoding framework, we use the *curiously recurring template pattern* (CRTP). The runtime effects of static over dynamic polymorphism are shown in Figure 1. The combination of Boost Hana, CRTP, and C++14's generic lambda expressions allows us to avoid typical type resolving patterns such as the visitor pattern or hardly maintainable switch/case statements.

*Hyrise's Execution Model:* The mentioned iterators cover both major reading access patterns: full sequential and point accesses. The basic execution model in Hyrise (with few exceptions for the query compilation engine) follows the principles used in most

[1]Boost Hana: https://boostorg.github.io/hana/

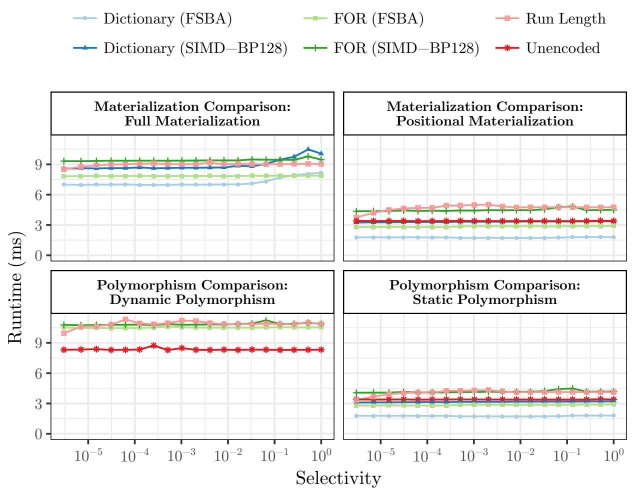

**Figure 1: Positional aggregation (i.e., aggregating 25% of the tuples via a given position list) on a vector of 1M integers. Top: impact of decompressing the full column upfront vs. positional accesses. Bottom: impact of dynamic polymorphism vs. static polymorphism for row accesses.**

modern columnar databases (cf. [1]). Predicates are descendingly ordered by their estimated costs and executed successively where each operator passes a list of qualifying positions to the next operator (i.e., position lists instead of materialized vectors). When parallel reads are preferable (e.g., as SIMD can be used), filters are executed in parallel and the results are intersected afterwards. Hence, the sequential iterator is typically used for the first filter predicate, while the following operators (e.g., following filters, joins, and aggregates) use the point-access iterator.

## 3 ESTIMATING PERFORMANCE

In order to decide for one compression technique over another, the runtime performance as well as the resulting storage requirements need to be estimated. Estimating runtime of a particular action is implemented in any database that optimizes incoming queries and needs to decide on the order of actions to some extent. However, we think that existing approaches are by far too inaccurate for our goals. The reason is that most databases do not optimize for the accuracy of each runtime prediction they make, but rather optimize to correctly estimate *the order of alternative decisions* to take. As soon as the order is sufficiently accurate, better models with smaller errors do not provide any further advantages. We argue, however, that for any reasonable decision on the physical database design, both the potential advantages (here, reduction of allocated memory) as well as the drawbacks (here, potentially increased runtimes) need to be known upfront.

While storage requirements are rather straightforward to estimate for most compression techniques (assuming knowledge about, e.g., the row count and the number of distinct elements), we found manually crafted cost models for runtime predictions to be problematic. Due to the various CPU and compiler optimization techniques on modern platforms (e.g., out of order execution, branch prediction, code reordering) manually crafted runtime models often turn out to be too inaccurate and cumbersome.

To accurately estimate runtimes nonetheless, we create an array of regression models for each compression technique (e.g., for different data types). We measured the runtimes of sequential as well as random accesses to compressed data structures. Note that we do not estimate the compression runtime or write performance of encoded schemes for two reasons. First, Hyrise ensures that mutable chunks are never compressed but only immutable chunks are. Hence, no writes to immutable chunks occur apart from MVCC modifications. Second, the decoding of segment columns for reading happens significantly more often than encoding. Thus, we consider the compression costs to be amortized soon after anyways and ignore them.

We evaluated three established regression methods: gradient-boosted regression trees (in our case, XGBoost [7]) and two linear regressions with different minimization objectives. One linear regression variant uses ordinary least squares (OLS) and the other has been adapted to use a relative (normalized) error metric. The reason to use a model minimizing relative errors is that non-normalizing models are less suitable in our case as they tend to optimize the prediction of long runtimes.

Typically, the runtimes of database operators are, however, heteroscedastic, meaning that the variance differs significantly for varying input parameters (e.g., the selectivity or the table size). As a result, when estimating runtimes of operations with very short expected runtimes (e.g., a scan operation on a small dimension table with a low selectivity), extrapolating models often estimate negative runtimes. In our case, a relative error metric is a better fit as estimation errors are equally important on short and long running operations.

Table 1 shows the error rates on the example of the dictionary-encoded model for integer values. We evaluated three data sets (each set has previously been split for training and testing), one including all measurements, one including measurements below the median runtime, and one including measurements equal to or above the median runtime. For typical regression metrics, such as the *mean squared error* (MSE), XGBoost shows the best results for all data sets. However, looking at the relative error metrics *mean average percentage error* (MAPE) and *Ln Q* [15] shows that the adapted linear regression has a lower error on two of the three data sets. In Hyrise, we consider relative error metrics to be superior as the resulting model's applicability increases. On top of determining accurate rankings during access path selection, it can also be used to estimate the runtime of complex queries (e.g., queries containing nested queries which are short running but executed many times).

| Quartiles | Metric | Linear Regression | | XGBoost |
|---|---|---|---|---|
| | | (OLS) | (adapted) | |
| $(Q1 - Q4)$ | MSE | 498 | 499 | 323 |
| | MAPE | 1.18 | 1.03 | 2.07 |
| | Ln Q | 0.000 250 | 0.000 172 | 0.001 72 |
| $(Q1, Q2)$ | MSE | 3.34 | 3.34 | 2.80 |
| | MAPE | 1.39 | 1.10 | 3.29 |
| | Ln Q | 0.000 352 | 0.000 196 | 0.003 35 |
| $(Q3, Q4)$ | MSE | 993 | 995 | 643 |
| | MAPE | 0.971 | 0.969 | 0.855 |
| | Ln Q | 0.000 148 | 0.000 148 | 0.000 099 0 |

**Table 1: Comparison of three regression models and varying error metrics (model for dictionary-encoded columns storing integers; green showing the best model).**

As a consequence, we decided for the adapted linear regression model as it yields lower estimation errors and has two further advantages over more sophisticated approaches such as gradient-boosted trees or net-based approaches. First, most tree-based approaches do not support inter- and extrapolating predictions of out of sample values, which regularly happens in our scenario. Second, both learning as well as predicting of linear models is efficient, fast, and can be implemented without additional dependencies in a comparably short time frame.

While the current approach to estimate runtimes is suitable for the scenario described here, i.e., finding the best configuration for a given memory budget, it is not sufficiently covering other important scenarios. Besides storage constraints, one common constraint is limiting the expected end-to-end runtime for a given workload to be at most $n\%$ larger than the runtime with the current configuration. Think of cloud scenarios where a database system can be redeployed on another (potentially virtualized) server at any time. A given workload in this example can consist of a set of queries with tight runtime constraints (e.g., caused by existing service level agreements).

## 4 COMPRESSION SELECTION

Selecting a suitable compression configuration requires knowledge about the particular data characteristics as well as the workload. Most current databases use simple heuristics to choose a compression scheme for a given column. However, those approaches neglect three major issues that we deem crucial: Compression configurations ought to be (i) adaptable with respect to a given memory budget, (ii) consider its impact on the decisions the query optimizer is going to make given the configuration, and (iii) consider opportunity costs usually exploitable in real-world systems and workloads.

While cost considerations always played an important role in database systems, the current trend towards self-adapting cloud systems emphasizes the need to reduce the memory consumption (amongst others) as much as possible while ensuring acceptable performance. In fact, a large database vendor told us that for many cloud installations, TCO (total cost of ownership) reductions are a far more pressing issue than performance.

As such, the database must understand the performance impact of varying memory budgets. It needs to understand the interplay between space consumption and performance, e.g., to apply maximum compression to a table that is virtually never accessed. At the same time, such a memory budget-driven system should degrade gracefully for decreasing budgets.

Our selection framework accepts a given memory budget that should not be exceeded. The system uses heavier compression schemes for data that is rarely accessed or whose access patterns do not suffer from heavy compression, while frequently accessed tables might use light-weight compression schemes or no compression at all. With the workload at hand, it might make sense to lose certain performance by applying heavier compression for a less often accessed table and invest that gained space to frequently accessed tables (cf. opportunity costs).

### 4.1 Greedy Selection Heuristics

The goal of compression selection is to determine a compression configuration for a given data set and workload. The selected configurations should gracefully degrade for decreasing memory budgets. The memory consumption of the resulting compression configuration ought to be within the given memory budget. We evaluated two heuristics and static configurations for a synthetically generated CH-benCHmark-like workload.

To gather workload information, Hyrise parses the database's query plan cache. This information is fed to the selection heuristics and includes for each physical column, e.g., which operations are executed with which access path taken (e.g., full linear accesses or random probing accesses).

We implemented two greedy heuristics which have been proposed for a related field of physical design optimization: index selection. As the first heuristic, we implemented a density-based heuristic that chooses the first compression technique from all applicable techniques ordered ascendingly by their size-to-performance improvement ratio, comparable to [16]. As the second heuristic, we implemented a performance-greedy heuristic that selects compression schemes ordered ascendingly by their expected performance improvement, comparable to [11].

The results are shown in Figure 2. We make two observations. First, the "All FOR (FSBA)" configuration has shown to be a good trade-off between performance and memory consumption. No heuristic was able to find a comparable configuration as FOR encoding is often neglected by both greedy heuristics. Second, simple greedy heuristics are not sufficient as they (i) fall short in covering the whole range of permitted memory budgets and (ii) can even be outperformed by simple static heuristics.

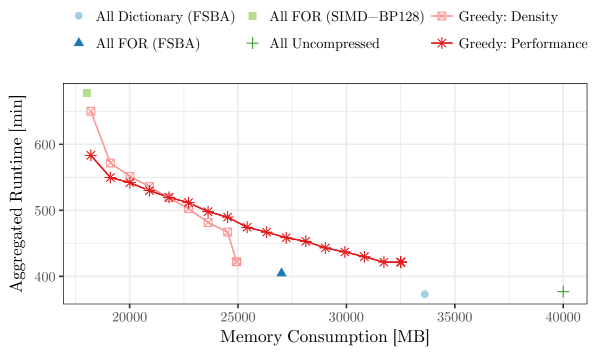

**Figure 2: Comparison of static compression configurations and budget-driven heuristics.**

The reason is that simple heuristics do not incorporate interactions affecting the optimization of query plans (cf. *index interaction*). One simple example is the effect of ordering conjunctive filter chains when the predicate with the lowest selectivity (hence, preferably being executed first) happens to be heavily compressed and hence slow to access. Optimizers using non-logical cost models – as done in Hyrise – might yield completely different query plans. To cover such cases, we think more elaborate selection approaches (e.g., recursive approaches as [3] and [6], or ILP-based approaches as [9]) are a necessary next step.

### 4.2 Robustness

The robustness of compression configurations is another crucial aspect. The more a segment is compressed as the expected workload is infrequently or not accessing it at all, the more expensive this decision might turn out when workloads shift.

To provide robust configurations, we use a simple framework to generate additional workloads which are evaluated together with the actually provided workload. Instead of selecting the compression configuration with the lowest runtime for the given workload, we choose the configuration minimizing the aggregated runtime of all workloads.

The creation of alternative workloads is done by shuffling and adding queries. First, for every query being part of the provided workload, we randomly select a new execution count based on the normal distribution around the actual execution count. Second, to counter the problem of heavy-weight compression for infrequently accessed segments, we manually add a query for each table to the workload that projects all columns. Both the number of generated workloads as well as the number of executions for the manually added query are configurable.

## 5 RELATED WORK

The object of integrating and selecting compression schemes has been researched in several previous works. Abadi et al. presented their C-Store extension to support various compression schemes [2] allowing operations directly on compressed data. A unified interface allows exploiting several compression scheme-unique optimizations while some operations have to fall back to a virtual method call-based interface. Further, the authors proposed a decision tree for the selection of compression schemes which uses both workload and data properties, but which does not adapt to changing environments nor considers memory budgets.

Lemke et al. presented a performance-optimized approach for TREX, the predecessor to SAP HANA [14]. At the cost of code complexity and maintenance efforts, compression schemes are explicitly handled within most database operators allowing to fully exploit vectorization and other optimizations.

Lang et al. presented data blocks for HyPer [13]. The authors focus the interplay of vectorizing operation on compressed vectors and their query compilation engine. HyPer selects compression schemes based on data characteristics.

## 6 CONCLUSION

We presented the current state of column compression in Hyrise. We think that compression selection will be an increasingly crucial topic. Mostly caused by (i) an increasing autonomy of database systems adapting themselves to changing environments and (ii) the move from on-premise to cloud-based installations which emphasizes the need for reduced main memory footprints.

## REFERENCES

[1] D. Abadi, , P. A. Boncz, S. Harizopoulos, S. Idreos, and S. Madden. 2013. The Design and Implementation of Modern Column-Oriented Database Systems. *Foundations & Trends in Databases* 5, 3 (2013), 197–280.
[2] Daniel Abadi et al. 2006. Integrating compression and execution in column-oriented database systems. In *Proc. SIGMOD*. 671–682.
[3] Martin Boissier et al. 2018. Hybrid Data Layouts for Tiered HTAP Databases with Pareto-Optimal Data Placements. In *Proc. ICDE*. 209–220.
[4] Peter A. Boncz et al. 1999. Database Architecture Optimized for the New Bottleneck: Memory Access. In *Proc. VLDB*. 54–65.
[5] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. 2008. Breaking the memory wall in MonetDB. *Commun. ACM* 51, 12 (2008), 77–85.
[6] Surajit Chaudhuri and Vivek R. Narasayya. 1997. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *Proc. VLDB*. 146–155.
[7] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *KDD*. ACM, 785–794.
[8] Patrick Damme et al. 2017. Lightweight Data Compression Algorithms: An Experimental Survey (Experiments and Analyses). In *Proc. EDBT*. 72–83.
[9] Debabrata Dash et al. 2011. CoPhy: A Scalable, Portable, and Interactive Index Advisor for Large Workloads. *PVLDB* 4, 6 (2011), 362–372.
[10] M. Dreseler et al. 2019. Hyrise Re-engineered: An Extensible Database System for Research in Relational In-Memory Data Management. In *Proc. EDBT*.
[11] Hideaki Kimura, Vivek R. Narasayya, and Manoj Syamala. 2011. Compression Aware Physical Database Design. *PVLDB* 4, 10 (2011), 657–668.
[12] Tirthankar Lahiri et al. 2015. Oracle Database In-Memory: A dual format in-memory database. In *ICDE*. IEEE Computer Society, 1253–1258.
[13] Harald Lang et al. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In *SIGMOD*. 311–326.
[14] Christian Lemke et al. 2010. Speeding Up Queries in Column Stores - A Case for Compression. In *Proc. DAWAK*. 117–129.
[15] Chris Tofallis. 2015. A better measure of relative prediction accuracy for model selection and model estimation. *JORS* 66, 8 (2015), 1352–1362.
[16] Gary Valentin et al. 2000. DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. In *Proc. ICDE*. 101–110.
[17] Thomas Willhalm et al. 2009. SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units. *PVLDB* 2, 1 (2009), 385–394.