Redesigning Query Engines for White-box Compression

Diego Tomé
tome@cwi.nl
CWI Amsterdam, NL
Supervised by Peter Boncz

ABSTRACT

Modern columnar databases heavily use compression to reduce memory footprint and boost query execution. These techniques, however, are implemented as a "black box", since their decompression logic is hard-coded and part of the table scan infrastructure. We proposed a novel compression model called *White-box compression* that views compression actions as *functions* over the physical columns stored in a block. Because these functions become visible as expressions in the query plan, many more optimizations can be made by the database system, boosting query execution speed. These functions are *learnt* from the data and also allow the data to be stored much more compactly, by decomposing string values, storing data in appropriate data-types automatically, and exploiting correlations between columns.

White-box compression opens up a whole new set of research questions. We started with (1) How to learn white-box compression expressions (functions) from the data automatically? This Ph.D. research will subsequently study (2) How to leverage white-box compression with (run-time) query optimizations? (3) How can we integrate white-box compression in a query engine, if the white-box functions may be different for each block of data?

1. INTRODUCTION

Data compression is widely used on analytical databases to reduce data storage size, as well as data transfer sizes (over the network, disk, RAM) and provide faster query execution. This is often effective on columnar databases, where the data pertaining to the same column are stored contiguously because compression algorithms perform better on data with low information entropy [3]. While data transfer can benefit from the improved compression ratio in columnar databases, query execution might suffer from slow decompression, requiring careful consideration of which compression technique should be applied.

The literature provides a number of compression techniques for databases. On the one hand, there are general-purpose compression methods based on Huffman [12], or arithmetic coding [19], and Lempel Ziv [20]. While they achieve good compression ratios, encoding/decoding speeds are relatively low, typically impacting query performance. For this reason, columnar databases rely on compression methods that are more light-weight, such as Run Length Encoding (RLE), Frame-of-Reference (FOR), and Dictionary compression (DICT)[3]. These lightweight schemes take into account some knowledge of the data-type and -distribution, resulting also in good compression ratio but much higher (de)compression speed.

A limitation of the state-of-the-art is that these existing techniques are implemented as "black-box" in the current database systems and the decompression logic is hidden inside the scan. The query plan is not aware of any decompression step, while the current approach is to eagerly decompress all the data in the scan, keeping the execution engine oblivious of the compression techniques – wasting optimization opportunities like predicate evaluation over partially decompressed data. Furthermore, we observed that in real-life datasets, data is often encoded in wrong data types (typically as strings), contain codes that combine strings and numerical parts (to which lightweight compression is not applicable), and/or has highly correlated columns [9]. Regarding the latter, columnar formats like Parquet compress datasets with strong column correlations worse than row-formats such as Avro, because of the general-purpose compression typically slapped on top of these formats (in a row-format, the co-located redundancies in a row get compressed away). Column stores store each column independently and lose this opportunity.

With *white-box compression* we proposed a completely new framework for compression in database systems. A table is a set of *logical columns* that is reconstructed by applying data-dependent *functions* over so-called *physical columns* that are stored. The compression function is therefore also data (meta-data), stored in the block header. This function is learnt from the data when writing it into blocks. We showed in [9] that this is feasible, that it greatly reduces storage size, and provides interesting query optimization opportunities.

However, we argue that to make this new idea usable in data systems, we need to explore new query optimization opportunities and redesign the query engine. Not only does white-box compression introduce unexplored opportunities for selection push-down, late decompression, and compressed execution. A significant challenge introduced by white-box compression is that when a block of data is

Proceedings of the VLDB 2020 PhD Workshop, August 31st, 2020. Tokyo, Japan.

written to disk, the white-box compressed model is learnt, depending on the characterization of that block of data. As such, each block may use a (slightly) different compressed representation. This already used to be the case in traditional compression, but since black-box compression hides the compression from the query engine this is only felt in the scan. With white-box compression, the compression functions, which are computational query plan expressions[1] will change continuously during query execution, whenever data from a new block is processed. This calls for a database system that *continuously re-optimizes its query plans and adapts them to the current characteristics of the data.*

Paper Structure. The rest of this paper is structured as follows. Section 2 provides an overview of related work. Then, in section 3, we describe the white-box compression model and we discuss possible solutions for the challenges that arise. Finally, in section 4, we discuss our research plan.

2. RELATED WORK

The problem of efficient compression on database systems has received significant attention over the years [10, 16, 21, 3]. As a result, compression has been well-explored on columnar databases for efficient query processing [21, 3, 5, 11, 15, 8, 14]. On these databases, one can achieve a good balance between compression ratio and performance with lightweight techniques.

The lightweight techniques provided advances on exploiting compressed data during the data processing pipeline, the so-called compressed execution [3, 4]. As a result, It brought performance improvement by allowing the push-down of predicates to compressed data in the scan operator [18]. On more advanced analytical systems like Hyper [14], data is stored in self-contained blocks allowing predicate push-down and selective scan on compressed data. Nevertheless, they have to sacrifice storage by keeping a byte-addressable representation.

Decompression beyond scans has been also explored [6] by re-compressing the data in between operators. The goal is to reduce the memory footprint for large intermediates but it is limited to the column-at-time processing model. On white-box compression, decompression is part of the query plan which allows the optimizer to delay decompression and push-down predicates to partially decompressed data.

Google recently described its query engine called Procella and its new big data file format Artus [7]. Artus introduces customized versions of lightweight techniques like RLE, Delta, and Dictionary encoding. On the API of Artus, RLE columns and dictionary indices are directly exposed to the query engine, allowing the engine to aggressively push computations down to the data format. In White-box compression, we also want to expose the compressed data to the query engine. Rather than implementing FOR explicitly, white-box compression sees FOR as an additional function, that adds a constant base to a physical column holding a small integer. RLE in white-box compression stores a logical column as two (much shorter) physical columns holding (count, value). On top of that, white-box compression allows much more complex data transformations, as well as the exploitation of column correlations (e.g. different dictionary-encoded columns using the same physical column holding the codes).

[1]In our current model, they could become even more adventurous as follow-up work.

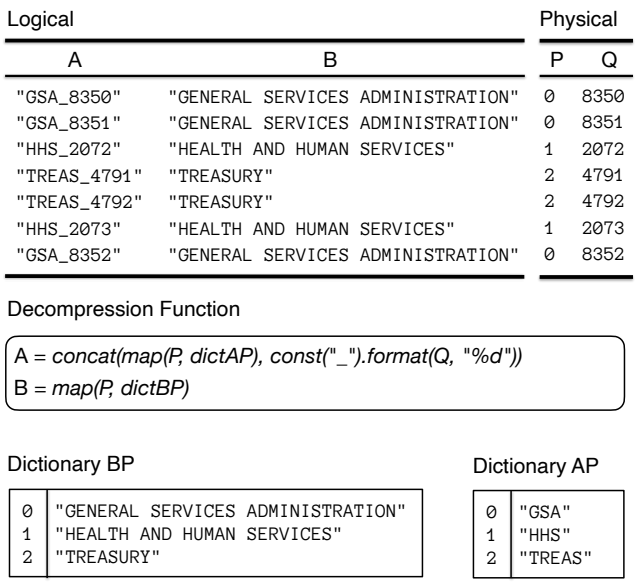

Figure 1: White-box compression model applied on logical columns A and B.

More recent research in compression has moved towards a storage method for decomposing string attributes in column stores [13]. This decomposition of strings relies on finding patterns to split the string into segments and compress them independently. White-box compression, on the other hand, is a more generic model, not restricted to strings, and which as mentioned can leverage column correlations. Further [13] restrict query execution optimizations to the scan library, while in our approach, compression functions become computational expressions part of the query plan. Making that work over blocks that are compressed differently is one main challenge addressed in this thesis.

3. WHITE-BOX COMPRESSION

In white-box compression, we define an operator as a function o that takes as input zero or more columns and optional metadata information and outputs a column: o: $[C \times C \times ...] \times [M] \rightarrow C$. The domain of o is composed of columns and metadata and the co-domain is a set of columns. A column is defined by its data type and the values that it contains. All the values that do not fit the chosen data representation are considered *exceptions* and receive special treatment. In the model, these values are stored separately in physical exception columns that can themselves be further white-box compressed.

Figure 1 illustrates a table compressed with white-box compression. Column A has a particular pattern composed of a string prefix, an underscore character, and a number. In this case, it is not possible to compress this column with dictionary encoding because it has a high cardinality. Therefore, the only way of compressing this column would be by using a heavyweight technique like LZ4 or some other variant that has the problem of slow de/compression.

Type/Column	Logical	Physical
varchar	80.3%	3.0%
tinyint	0%	31.7%
smallint	13.7%	60.4%
double	2.3%	0.3%
decimal	2.1%	4.6%
integer	0.9%	0%
boolean	0.7%	0%

Table 1: Data types distribution for logical and physical columns on the Public BI benchmark. Physical columns are the result of applying white-box compression.

For cases not covered by any lightweight technique, the white-box model can enable compression by changing the physical representation of columns. For instance, column A can be decomposed into three other columns that can be further compressed. The prefix string can be now stored as the dictionary AP, the underscore as a constant and the number can be stored as an integer that can be further compressed with some other lightweight technique.

Another compression opportunity happens when column B is stored in the dictionary BP. In this case, we are able to represent one column as a function of another (i.e. column correlation) by identifying the same association between dictionaries AP and BP. As a result, only one physical column (i.e. P) is stored together with the two dictionaries and the decompression function to reconstruct the logical columns.

Column Correlations. The particular focus of this work on correlations is based on their frequent occurrence on real-world datasets. We have recently introduced the Public BI benchmark [1], a real-world dataset derived from 46 of the biggest Tableau workbooks [2]. While exploring this data, we noticed that it tends to comprise patterns not found in synthetic database benchmarks like TPC-H and TPC-DS. In this dataset, data is often skewed in terms of value and frequency distribution and it is correlated across columns. In particular, most of the correlations are between nominal values (i.e. strings).

In our work [9], we formally define the white-box compression model and propose a learning algorithm to identify patterns on the data. Our initial approach already doubles the compression factor on the public BI benchmark [1]. White-box compression is very effective here because this data has many string columns. In Table 1 we show the data type distribution for logical and physical columns on the public BI benchmark. Thanks to white-box compression the volume of strings is reduced and the dataset becomes more compressible. Integer columns are stored in smaller types while boolean columns are represented as constant operations inside an expression. Correlated columns play an important role since we noticed a reduction of 70% in the number of columns after white-box compression.

4. WHITE-BOX DECOMPRESSION

In the previous section we showed how White-box compression is defined and the main advantages over black-box

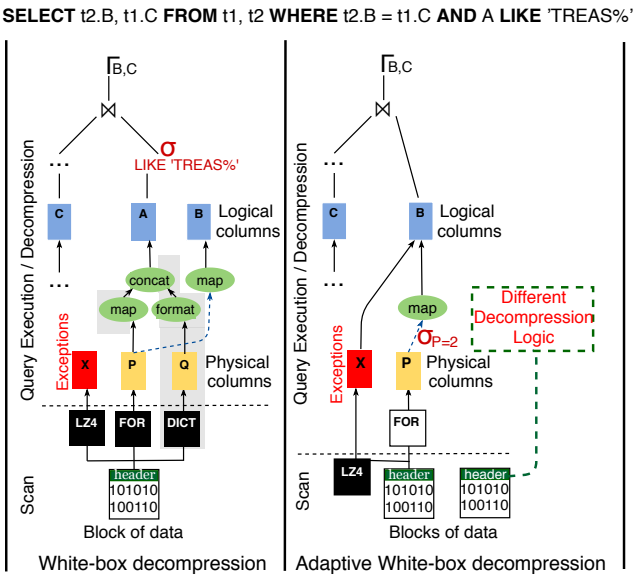

Figure 2: Fusing decompression and Query execution on top of white-box compressed data.

compression. We now discuss fast decompression and its integration in the query plan.

On the white-box model, the decompression functions to rebuild the logical columns are stored in the block metadata. Decompression will be implemented in two phases. In phase 1, the compressed data is first unpacked using fast SIMD codecs into byte-addressable physical columns. This partially decompressed representation can be represented as columnar vectors in the query plan as long as needed, and once an operator like a join, group by, or projection requires the logical representation the second phase of the decompression is performed (lazy decompression). Many operations can thus take place on the data still in (partially) compressed form. The second phase is the full decompression of physical columns into logical columns.

Figure 2 depicts our proposal for decompression and query execution on a white-box representation. In the left part, we show an unoptimized approach where the physical columns are black-box decompressed into columns X, P, and Q. In this scenario, the query optimizer is able to push-down predicates straight to physical columns, avoiding unnecessary decompression steps. The challenge in this approach is the black-box decompression steps still present, which limits the ability of the execution engine to operate over the physical columns in compressed format.

In the right part of Figure 2, we illustrate the optimized version of the query plan fused with decompression. The optimizer can adapt the plan by pruning some decompression steps and pushing-down predicates to the partially decompressed data. The column compressed with frame-of-reference becomes also a white-box representation represented by a column of values and the reference with difference operator.

Adaptive Query Processing. In [9], we considered a simple approach in which the entire logical column would use the same decompression function for the whole table. If these functions may change for each block of data, integration of white-box compression in the query engine becomes

more tricky: whenever a block of data is read, we should stop the query execution and re-instantiate a new query plan and re-optimize. We argue that a vectorized engine is more likely to succeed in quickly handling such changes, than an alternative approach with a JIT-compiled engine, which would introduce-recompilation latency for each new data block. To further save time, we propose to split query optimization into a main *strategical* phase that is executed once before execution and perform lightweight *tactical* re-optimization whenever a new data block is brought in and the strategic query execution plan is adapted to it.

5. RESEARCH PLAN

We believe *white-box compression* is the foundational idea for a next-generation of database engines, where compressed columnar execution is critical to leverage powerful SIMD units. It also unlocks many optimization possibilities by *learning* data representations from the data and is less vulnerable to ill-designed database layouts that are often observed in cloud usage situations.

We describe the following as the aspects that shall be explored in this research during the next two years:

(De)compression Library: We start with developing a basic library for compression and decompression of data in the white-box representation. Our first goal is to have this library independent of any database system and evaluate compression and decompression speeds.

Storage Layout: Besides the de/compression library it is necessary to define how expression trees will be stored and instantiated during query execution. Therefore, the second step is the definition of a file layout to represent data on white-box representation on disk. We plan to have all the information for white-box decompression on block headers that will be instantiated whenever a block is loaded.

Compressed Execution: In a white-box compression model, the push-down of database operators within the decompression tree becomes more transparent. It is not so clear, however, which kind of expressions can be built aiming compressed execution or which database operators allow such execution. To clarify these question we will perform a careful investigation on which database operators get benefit from compressed execution and how to generate decompression expressions that enable such an approach.

Vectorization vs. JIT Compilation: With an adaptive query processing on white-box representation a JIT-compiled engine might suffer from compilation overhead. For every new block, the decoder has to be JIT-compiled and the overhead grows with the number of different compression schemes per column. Therefore, on this thesis, we narrow down our design to a vectorized query engine where changes in the decompression logic can be best handled and interpreted. We will consider DuckDB [17] as our target since its the only open-source columnar store with a vectorized execution engine.

Tactical Query Optimization: Compressed execution offers opportunities to better use SIMD resources (thinner data can execute in more lanes in parallel), but also allows for early pruning of data using cheap(er) test on thin fragments of the data. It can also lead to hash-tables that are smaller and thus faster and network communications that are reduced. However, in order to apply these optimizations in the face of continuously changing white-box compression functions, we need to quickly re-optimize query plans when new data arrives. For this purpose, we plan to split query optimization into two phases: a heavy strategical phase that includes join ordering and is executed only once, and a tactical phase that applies cheap optimization that leverage compressed execution opportunities, specifically.

6. REFERENCES

[1] Public BI Benchmark. https://github.com/cwida/public_bi_benchmark.

[2] Tableau Public. https://public.tableau.com.

[3] D. Abadi, S. Madden, and M. Ferreira. Integrating Compression and Execution in Column-oriented Database Systems. In *SIGMOD*, pages 671–682, 2006.

[4] D. J. Abadi. *Query Execution in Column-oriented Database Systems*. PhD thesis, 2008. AAI0820132.

[5] C. Binnig, S. Hildenbrand, and F. Färber. Dictionary-based Order-preserving String Compression for Main Memory Column Stores. In *SIGMOD*, pages 283–296, 2009.

[6] P. Damme, A. Ungethüm, J. Pietrzyk, A. Krause, D. Habich, and W. Lehner. Morphstore: Analytical query engine with a holistic compression-enabled processing model. *arXiv preprint arXiv:2004.09350*, 2020.

[7] B. C. et al. Procella: Unifying Serving and Analytical Data at YouTube. *PVLDB*, pages 2022–2034, 2019.

[8] Z. Feng, E. Lo, B. Kao, and W. Xu. ByteSlice: Pushing the Envelop of Main Memory Data Processing with a New Storage Layout. In *SIGMOD*, pages 31–46, 2015.

[9] B. Ghita, D. G. Tomé, and P. A. Boncz. White-box Compression: Learning and Exploiting Compact Table Representations. In *CIDR*, 2020.

[10] G. Graefe and L. D. Shapiro. Data Compression and Database Performance. In *Symposium on Applied Computing*, pages 22–27, April 1991.

[11] B. Hentschel, M. S. Kester, and S. Idreos. Column Sketches: A Scan Accelerator for Rapid and Robust Predicate Evaluation. In *SIGMOD*, pages 857–872, 2018.

[12] D. A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *IRE*, pages 1098–1101, 1952.

[13] H. Jiang, C. Liu, Q. Jin, J. Paparrizos, and A. J. Elmore. PIDS: Attribute Decomposition for Improved Compression and Query Performance in Columnar Storage. *PVLDB*, 2020.

[14] H. Lang, T. Mühlbauer, F. Funke, P. A. Boncz, T. Neumann, and A. Kemper. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage Using Both Vectorization and Compilation. In *SIGMOD*, pages 311–326, 2016.

[15] Y. Li and J. M. Patel. BitWeaving: fast scans for main memory data processing. In *SIGMOD*, pages 289–300, 2013.

[16] O. Polychroniou and K. A. Ross. Efficient Lightweight Compression Alongside Fast Scans. In *DAMON@SIGMOD*, pages 9:1–9:6, 2015.

[17] M. Raasveldt and H. Mühleisen. DuckDB: An Embeddable Analytical Database. In *SIGMOD*, page 1981–1984, 2019.

[18] V. e. a. Raman. DB2 with BLU Acceleration: So Much More Than Just a Column Store. *PVLDB*, pages 1080–1091, 2013.

[19] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic Coding for Data Compression. *Commun. ACM*, pages 520–540, 1987.

[20] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, pages 337–343, 1977.

[21] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar RAM-CPU Cache Compression. In *ICDE*, pages 59–, 2006.