White-box Compression: Learning and Exploiting Compact Table Representations

Bogdan Ghiță
CWI Amsterdam, NL

Diego Tomé
CWI Amsterdam, NL

Peter Boncz
CWI Amsterdam, NL

ABSTRACT

We formulate a conceptual model for *white-box compression*, which represents the *logical* columns in tabular data as an openly defined function over some actually stored *physical* columns. Each block of data should thus go accompanied by a header that describes this functional mapping. Because these compression functions are openly defined, database systems can exploit them using query optimization and during execution, enabling e.g. better filter predicate push-down. In addition, we show that white-box compression is able to identify a broad variety of new opportunities for compression, leading to much better compression factors. These opportunities are identified using an *automatic learning* process that learns the functions from the data. We provide a recursive pattern-driven algorithm for such learning. Finally, we demonstrate the effectiveness of white-box compression on a new benchmark we contribute hereby: the *Public BI benchmark* provides a rich set of real-world datasets.

We believe our basic prototype for white-box compression opens the way for future research into transparent compressed data representations on the one hand and database system architectures that can efficiently exploit these on the other, and should be seen as another step into the direction of data management systems that are self-learning and optimize themselves for the data they are deployed on.

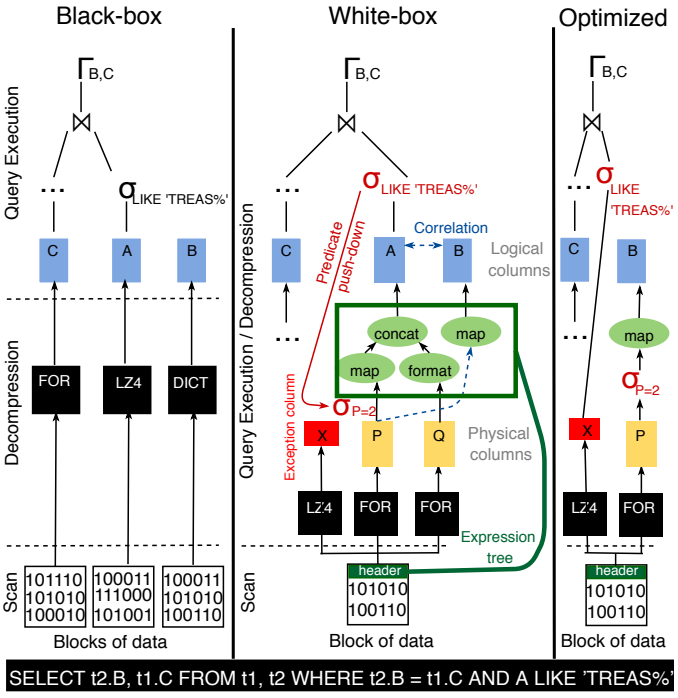

Figure 1: In white-box compression, compressed blocks in their header contain expressions that define how the *logical* columns of a table are computed as a function of the stored *physical* columns. **Please skip to Table 2 to see the example data and compression functions.** In traditional – "black-box" – compression methods, decompression is opaque to the query engine. White-box compression unlocks query optimizations such as predicate push-down, and physical column pruning, and achieves better compression, e.g. by exploiting a *correlation* between columns A and B. The compression functions are ***learned* from the data.**

1. INTRODUCTION

Data compression is an important technique for analytical data management. Apart from reducing data storage cost, it reduces data transfer sizes and this can also speed up query execution, because smaller memory-, disk - and/or network-accesses take less time. Compression is especially effective in columnar storage, that stores data pertaining to the same distribution (i.e. column) together, and is widely used in popular columnar file formats such as ORC and Parquet. On the one hand, there are general-purpose compression schemes such as Huffman [10], or arithmetic coding [23], and a number of Lempel Ziv [24] variants. Even though some of these trade compression ratio for higher speed, such as LZ4 and Snappy, we call these "heavy-weight" methods, because encoding/decoding speeds are relatively low, typically impacting query performance. On the other hand, there are much faster "light-weight" compression schemes that need knowledge of the data-type and -distribution, such as Frame-of-Reference, RLE and Dictionary compression [3, 25].

We call all these existing techniques *black-box* compression, because their decompression logic is hard-coded and

10th Annual Conference on Innovative Data Systems Research (CIDR '20) January 12-15, 2020 , Amsterdam, Netherlands.

the query operators in the database system cannot directly operate on the their compressed bit representations. In this paper, we propose *white-box* compression that not only makes (de-)compression transparent and optimizable, but also is able to strongly reduce the size of real-world data.

Real world datasets tend to present characteristics that do not occur in synthetic benchmarks like TPC-H [4] and TPC-DS [17]. Not only is data often skewed in terms of value and frequency distribution, but it is also correlated across columns. Columns, or parts of columns, are often not stored in the most appropriate data type (e.g. sometimes almost numeric columns end up as strings). The "Get Real.." DBtest paper [22] highlighted the lack of realism in the datasets generated for existing benchmarks, but did not release the data it described. As part of this project, we first created a new benchmark: the Public BI benchmark [1], based on 46 of the largest Tableau Public workbooks. We hope to have contributed a useful resource for database research with it. In our case, we used its rich collection of real-world data for inspiration and evaluation, to identify and characterize new methods for compression and providing better data representations.

In recent years, there have been advances that try to unify the design space of e.g. index structures, so a specific data structure can be generated, that is optimal for a particular workload [11]. Even more broadly, there are initiatives for so-called *instance-optimized* data management systems [13], that optimize the full system architecture for the workload, often using machine learning (ML) techniques. Our work on white-box compression envisions future data formats that are also instance-optimized. We propose to see compression as a transparently described function over the data. Thus, in a white-box compressed data format, the block header in a data file does not only contain basic information about the columns of data in it (data types, offsets, etc), but also contains the *description* of the compression function that the decompressor needs to instantiate and execute for decompressing the data.

In the following, we quickly walk through the main research questions this raises. We investigate the first three in resp. Section 2, 3 and 4. The latter two we did not investigate yet, but all of them provide future research opportunities for the data management community.

What could these functions look like? The function should have the appropriate *expressive power* to handle patterns that occur in real-world data. Also, the *speed* of (de) compression should remain high, such that compressed data can be accessed fast. While one can even think of using ML models as functions, the white-box compression model we define in Section 2 starts with functions that are standard column expressions (trees of operations on columns or constants) that query engines already support.

How does the system *learn* these functions during compression? What is the best function to represent a column, depends foremost on the data, but also the workload could be taken into account. In Section 3, we define finding a good function as an optimization problem and provide a rule-based algorithm that splits *logical* columns to store the data in *physical* columns that are more (black-box) compressible; and subsequently tries to find correlations between physical columns to reduce duplicated storage.

How much can compression rate be improved? In Section 4, we evaluate our implementation of white-box compression on our Public BI Benchmark. We find that a factor 2 of storage space can be saved already, even though the methods we use are still rather simple. We speculate that compression rates in the future could be boosted even further, by more advanced techniques or even machine learning.

How can these functions be exploited in query optimization and execution? Since (de)compression is a transparent first phase of query execution, compression operators could be fused with query operators at runtime or might even cancel out. For instance, if a date column was stored as string and most queries start by casting it to date, this expensive operation can be skipped. Figure 1 highlights another prolific opportunity: white-box compression enabling selection push-down (e.g., by storing the column as a proper date, rather than as a string, a date-range condition on the casted string, can now be pushed).

How can data management systems quickly parse and execute such functions? Each block of data may come with its own functions. One could for instance use JIT-compilation, but would have to take counter-measures in order to contain compilation latency, which might be incurred for each block of data. Alternatively, a vectorized engine could be used, which does not have the latency problem. In this paper we use VectorWise as our experimental platform, but we defer research into efficient execution of white-box compression to future work.

2. WHITEBOX COMPRESSION MODEL

The white-box compression model represents *logical* columns as composite functions of *physical* columns. Logical columns are the columns as defined by the database schema, containing the tabular structure that the user expects to see. Physical columns are what we actually store on disk. There may be fewer (or more) physical columns than logical columns, and their data types may be different. In this initial approach to white-box compression, these functions are expressions that consist of simple, scalar, *operators*. While in our current work we use the same mapping between logical and physical columns for the whole table, white-box compression could in principle use a different mapping for different horizontal pieces (data blocks) of the table.

Formally, we define an operator as a function o that takes as input zero or more columns and optional metadata information and outputs a column: $o\colon [C \times C \times ...] \times [M] \rightarrow C$. The domain of o is composed of columns and metadata and the codomain is a set of columns. A column is defined by its data type and the values that it contains. The metadata is structured information of any type. Table 1 shows as a small set of operators we defined based on specific compression opportunities.

We can for instance represent a logical string column c_a as a physical integer column c_b using a formatting operator m_{format}. E.g. `"-12000"` $= format($`-12000`$,$ `"%d"`$)$, where

Detector	Operator		
Different Alphabet Zones	$concat$:	$C \times C \times [...] \rightarrow C$	$c = concat(c_a, c_b, [...])$
Numerics in String	$format$:	$C \times M \rightarrow C$	$c = format(c_a, m_{format})$
Few Unique Values	map:	$C \times M \rightarrow C$	$c = map(c_a, m_{map})$
Mostly Constant	$const$:	$M \rightarrow C$	$c = const(m_{const})$

Table 1: A few typical operators used in column expressions.

v_b = `-12000` and m_{format} = `"%d"`. The direct mapping representation of a column c_a as an integer column c_b through the mapping m_{map} is a key-value lookup in an array-like dictionary. E.g. `"TREAS"` = $dict_{AP}[2]$, where $dict_{AP}$ is an array of size 3 with values [`"GSA"`, `"HHS"`, `"TREAS"`]. The constant representation of a column indicates that all its values are equal to the constant value m_{const}. These operators can be composed, resulting in operator expressions or *expression trees*—tree structures with logical columns as root nodes, operators as internal nodes and physical columns as leaf nodes. Table 2 illustrates white-box compression on a small data sample from the Public BI benchmark. The logical columns A and B can be represented as composite functions of the physical columns P and Q, through the following expressions:

$$A = concat(map(P, dict_{AP}), const(\texttt{"\_"}), format(Q, \texttt{"\%d"}))$$
$$B = map(P, dict_{BP})$$

A	B	P	Q
"GSA_8350"	"GENERAL SERVICES ADMINISTRATION"	0	8350
"GSA_8351"	"GENERAL SERVICES ADMINISTRATION"	0	8351
"HHS_2072"	"HEALTH AND HUMAN SERVICES"	1	2072
"TREAS_4791"	"TREASURY"	2	4791
"TREAS_4792"	"TREASURY"	2	4792
"HHS_2073"	"HEALTH AND HUMAN SERVICES"	1	2073
"GSA_8352"	"GENERAL SERVICES ADMINISTRATION"	0	8352
Logical		Physical	

Table 2: Logical vs. physical data

We observe that column A has the following structure: a dictionary compressible prefix and a numeric suffix, separated by a the '_' character. If we store these logical parts separated into 3 columns C_{prefix}, C_{delim}, C_{suffix}, we can represent column A as their concatenation. Since C_{prefix} has repeated values, we can represent it more compactly as a mapping of column P—containing dictionary keys—and the dictionary $dict_{AP}$. We can represent C_{delim} through the *const* operator since all its values are equal to '_'. C_{suffix} contains numbers stored in strings. We can store these values more compactly as numbers, by changing the column data type. Therefore, we represent $C_{(suffix)}$ based on the numeric column Q, through the $format$ operator, with the format string `"%d"`. We move our attention to column B and observe that it is correlated with column C_{prefix}—and implicitly also to column P. We can therefore represent B as the mapping of column P and the dictionary $dict_{BP}$, without explicitly storing its values. In the end, we store only the physical columns P and Q and the metadata: $dict_{AP}$, $dict_{BP}$ and the constant string `"_"`.

Through these expressions we are able to store the data on columns A and B more compactly, by removing redundancy and using optimal data types, allowing further compression of columns P and Q with existing black-box numeric compression schemes (e.g. FOR). Note that a system with traditional black-box compression can only compress the high-cardinality string column A with slow LZ4, whereas the low-cardinality string column B could be stored with DICT. With white-box compression, logical column A now gets stored as two small integer columns P and Q, both highly compressible with the fast FOR scheme. Column B does not even require any storage anymore, as it is correlated with A and can be reconstructed from P.

Additionally, the same expressions create opportunities for faster query execution. Recall the example query from Figure 1: `select B,C from T1,T2 where B=C and A like 'TREAS%'`. A query execution engine that is aware of the data representation will filter the results by only fully reading the physical column P from $T1$, pushing down condition $P = 2$, and then generate both logical columns A and B based on P and Q, only for the qualifying tuples (i.e. potentially skipping many rows). However, given that compression and query execution are now fused, an optimizer could also perform *physical column pruning* and remove the whole calculation of A, since it is no longer needed (see Figure 1, rightmost side).

An additional and important part of white-box compression is *exception handling*. Exceptions are outlier values that do not fit the data representation (e.g. a value on column A that does not have the *prefix-delim-suffix* structure). We designed the model such that these values are stored separately in physical *exception columns*. These exception columns can then be recursively white-box compressed, thanks to the generic nature of the model. In our example query, we saw that predicate push-down gets rid of most of the string matching effort. However, we still need to do this more expensive check on the (few) rows where the compression function does not fit the data (in that case P holds NULL), and the original string is stored in exception column X (which holds NULL for data that fits). We simplified the optimized query on the right of Figure 1 a bit: the full pushed-down condition is P IS NULL or P=2 and the top-level filter condition is P IS NOT NULL or X LIKE 'TREAS%'.

We have seen how the four operators described above can be used to represent data more compactly and how full decompression can be postponed during query execution. However, the white-box compression model does not limit itself to these four operators. It is a generic model and supports any type of column operators (e.g. mathematical operators like addition or multiplication). Given the multitude of different possible representations of the same logical columns, a major challenge is exploring them and choosing the most suitable one. We approach the challenge of automatically learning patterns in the data and its representation in the next section.

3. LEARNING COMPRESSION

This section introduces the general problem of automatically learning white-box functions from a sample of data and generating expression trees. We further propose a learning algorithm and describe our proof-of-concept implementation.

3.1 Optimization problem

We define the compression learning process as an optimization problem: given a sample ($\mathcal{D}_s$) of a dataset ($\mathcal{D}$), its schema ($\mathcal{S}$), a set of operators ($\mathcal{O}$) and a cost model ($\mathcal{C}$), output a combination of operators in the form of an expression tree ($\mathcal{T}$) that, when applied to the full dataset, produces a minimum cost representation of it. The sample $\mathcal{D}_s$ is a representative subset of rows from $\mathcal{D}$. The schema $\mathcal{S}$ describes the structure of $\mathcal{D}$ in terms of columns and their data types. The operators $\mathcal{O}$ are functions that apply elementary transformations on columns, as described in the previous section. The cost model $\mathcal{C}$ provides a score which measures the effect

of representing $\mathcal{D}$ through the expression tree $\mathcal{T}$ in terms of two main metrics: 1) compressed size of $\mathcal{D}$ and 2) query execution time.

3.2 Learning algorithm

Algorithm 1 Exhaustive recursive learning

```
1: global O_list, M_cost                          ▷ operator list and cost model
2: function BUILDTREE(c_in, T_in)
3:     S_list ← empty solution list
4:     cost ← M_cost.evaluate(c_in, T_in)
5:     append solution (cost, T_in) to S_list
6:     for each o ∈ O_list do
7:         (cost, T_out) ← APPLYOPERATOR(o, c_in, T_in)
8:         append (cost, T_out) to S_list
9:     return min(S_list)                         ▷ solution with minimum cost
10: function APPLYOPERATOR(o, c_in, T_in)
11:    T_out ← copy(T_in)
12:    add operator o to T_out
13:    S_list ← empty solution list
14:    for each c_out ∈ output columns of o do
15:        (cost, T_c) ← BUILDTREE(c_out, T_out)
16:        append (cost, T_c) to S_list
17:    (cost, T_out) ← merge(T_out, S_list)       ▷ sum costs and merge trees
18:    return (cost, T_out)
```

We propose a recursive exhaustive algorithm for solving the compression learning optimization problem (described in pseudocode in Algorithm 1). It uses a predefined list of operators (O_{list}) and a cost model (M_{cost}). The algorithm takes as input a column (c_{in}) and an initially empty expression tree (T_{in})—the partial solution built so far—and outputs the best representation of c_{in} in the form of an expression tree (T_{out}) and its corresponding *cost*.

The algorithm uses the cost model to evaluate the cost of the current (partial) solution (i.e. not further representing the column through any operator). Then it compares it with the cost of the solutions resulted from applying each operator to the input column in a recursive process and chooses the solution with minimum cost. The algorithm terminates implicitly when no operator can be applied to the input column or when all operators that can be applied give no output columns. In practice, the dimension of the problem and algorithm termination can be controlled by imposing a maximum height for the expression tree. The complexity of the algorithm is $\mathcal{O}((o \times c_{out})^{h_{max}})$, where o is the number of operators, c_{out} is the average number of columns that an operator outputs and h_{max} is the maximum height of the expression tree that we allow.

3.3 Proof-of-concept implementation

We created a proof-of-concept implementation of the learning algorithm described above, in order to evaluate its feasibility in practice and to validate the white-box compression model. We briefly present the cost model and operators that we used, as well as the general architecture of the learning process, omitting implementation details due to space constraints. An in-depth description of our approach is presented in [6].

Cost model. We used a simple cost model based on a single metric: size of the physical data. We wanted to measure the impact of the white-box representation in terms of compression ratio, as well as its potential to create opportunities for better compression with existing methods. Therefore, we designed a cost model that estimates the size of a column as if it were compressed with existing lightweight compression schemes (RLE, FOR, DICT) or not compressed. Out of those estimated sizes, it outputs the smallest one, which is further used to compare solutions in the learning algorithm.

Operators and pattern detection. We used the operators listed in Table 1. These operators, however, are not applicable to every column, raising an additional challenge: matching columns with suitable operators based on their data type and properties. We addressed this challenge by implementing an automated pattern detection engine, which searches for patterns in each column and evaluates its compatibility with each operator. We created four pattern detectors which identify suitable columns for: 1) concatenation—by searching for columns that can be split based on their character set sequences; 2) data type change (formatting)—by identifying columns with suboptimal data types; 3) direct mapping—by finding pairs of columns that are correlated to each other and can be represented through a mapping; 4) constant representation—by identifying columns that have (mostly) a single value.

Column correlations. An interesting additional aspect of our work is related to correlated columns. We focused on finding dependencies between nominal categorical columns, with the purpose of reducing redundancy by representing them as functions of each other. We do this by building correlation mappings (dictionaries) between every two columns, which store the dependencies between their values. The values that do not match the correlation map are marked as exceptions. We identify highly correlated columns based on our own correlation coefficient: $1 - exception\_ratio$—which gives similar results to existing statistical metrics for correlation between categorical variables: Cramer's V and Theil's U [6]. In our proof-of-concept implementation we applied this technique later in the learning process on the leaves of the expression tree (i.e. the physical columns), enabling us to remove physical columns if they correlate with another physical column. This turns our the compression functions from a set of n expression trees into a proper *directed graphs* with n roots.

Updates on the white-box representation. When dealing with updates on compressed data,there are multiple approaches to minimize the modification of previous compressed blocks. Microsoft SQL Server treats small point updates as a combination of a delete followed by an insert [5]. While Vectorwise relies on a positional delta tree (PDT) structure [9] that handles update by keeping a delta with the differential data in memory. On White-box compression, we envision two options regarding where the updates should be performed, one can either operate on the logical level or on the physical level. On the logical level, one can keep a PDT for updates and during a scan we first check the delta and merge the differences. On the Physical level, however, the update would be inserted on the exception column and marked as a modified tuple.

4. EVALUATION

We evaluated the proof-of-concept implementation against a real system with enhanced compression capabilities: VectorWise [26]. We aim at showing the improvement brought

by white-box compression as an enhancement of existing systems: an intermediate representation layer that remodels the data to create better compression opportunities for existing lightweight methods. For this reason, we first white-box compress the data offline using the algorithm described in Section 2. Then, we load the resulting physical columns into VectorWise, such that it uses its black-box compression methods—patched versions of well known lightweight compression methods: PDICT, PFOR, PFOR-DELTA [26]. This allows us to measure how much compression ratio improves thanks to white-box compression.

We performed our evaluation on the Public BI benchmark [1]. As this is a new and unknown benchmark, we shortly present some of its characteristics, followed by the preliminary results we obtained.

4.1 Public BI benchmark

The data used in our experiments is part of the Public BI benchmark, the first fully user-generated benchmark for database systems. The data and queries were generated by downloading the 46 biggest Tableau Public workbooks mentioned in [22], collecting the SQL queries that the workbook triggers on them, and then exporting the data (usually one or a few similar tables per workbooks) and curating data and queries to make them portable to relational database systems beyond Tableau/Hyper. The Public BI benchmark consists of 386GB of real data and 646 analytical queries [1]. We developed this benchmark and chose it for our evaluation because of its data distributions, diversity in content and the extended character set, which make it suitable for evaluating compression solutions. In order to better understand the properties of real data and to gain more insight about it, we manually searched for patterns—common ways that users represent the data—in order to find opportunities for more compact representations. In summary:

– **Empty/missing values that are not nulls**—e.g. empty quotes, whitespace characters;
– **Dirty Data**: even though most columns contain homogeneously distributed data, many columns have some values that do not conform to the norm, or consist of multiple kinds of data that conform to different norms.
– **Leading/trailing whitespace**, some with the purpose of ensuring a common length of the values on `VARCHAR` columns;
– **Numbers and dates stored** in `VARCHAR` columns;
– **Strings with fixed structure composed of substrings from different distributions**—e.g. emails, urls, strings starting with a constant and ending in a number;
– **Correlations between columns**—mostly as categorical variables, but also numeric correlations.

The patterns mentioned above are very frequent in the benchmark data. One could conclude that database end-users often make sub-optimal data representation choices, in terms of database performance – this is both because they are not database performance experts, and are more interested in their business problem than in performance. We expect this situation to become even more frequent as databases move to the cloud and DBAs are no longer available. These sub-optimal data representations typically lead to extra query processing effort [2] and larger-than-necessary storage. With white-box compression, we can automatically optimize physical database storage and execution on user data, despite its logical sub-optimal form.

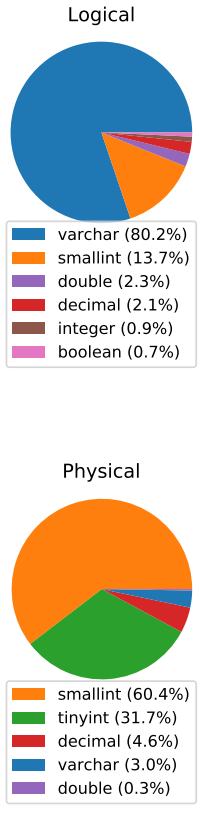

Figure 2: String volume in the new Public BI Benchmark reduces from over 80% to just 3% thanks to white-box compression.

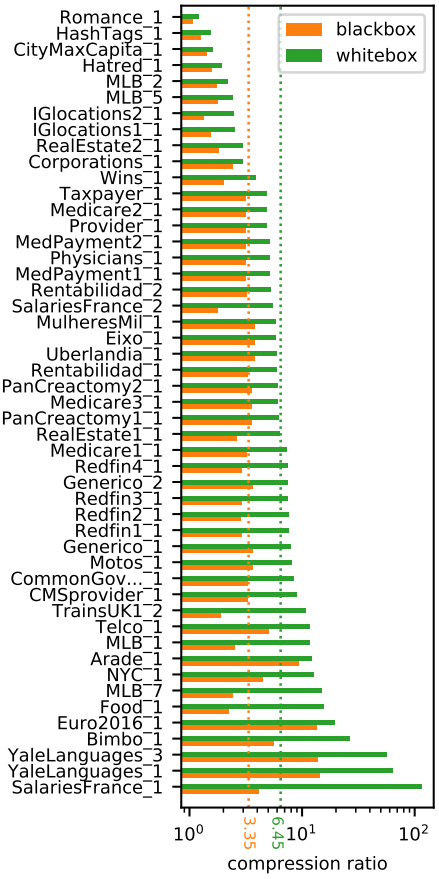

Figure 3: The overall compression factor achieved doubles to 6.19 thanks to white-box compression.

4.2 Experimental Results

We present some of the results of our evaluation on 49 tables from the Public BI benchmark, which we selected based on the observation that tables with the same schema have very similar data and sometimes are almost identical. We only kept one table for each unique schema and also removed very small tables (e.g. <1MB).

In Figure 3 we show the main results of our evaluation: the compression ratios of each table, considering the columns selected for white-box representation by the learning algorithm (which make up 68% of the size of the full tables). For most of the tables, the ratio achieved by white-box compression is significantly higher than the one of black-box methods. The rest of the tables, which have similar compression ratios, either contain less white-box compression opportunities or part of their data is already compressible with existing techniques. Overall, white-box compression brings an improvement of 1.92× over black-box compression in terms of the aggregated compression ratio on all tables (6.45 vs. 3.35). A major role in these results is played by column correlation, which makes up 28% of the operators in the

expression trees, leading to a smaller number of physical columns than logical columns (excluding exception columns). Moreover, the distribution of data types across these columns is also different: the majority of logical columns are `VARCHAR`, while the physical columns are mostly numeric—making data more compact and suitable for compression with existing black-box techniques (Figure 2). In terms of size, the 77.2GB of logical data is represented through only 3.7GB of compressed physical data and, interestingly, 11.3GB exceptions (0.16 average exception ratio). We are, therefore, able to compactly store 65.9GB (the non-exceptions) with a compression ratio of 17.8. The high share of volume for exception columns suggests future work towards reducing the number of exceptions or better compressing them.

We conclude that our learning algorithm effectively identifies opportunities for more compact data representations, significantly improving compression ratios.

5. RELATED WORK

Existing work [3, 25, 14, 18, 8] on database compression covers a wide range of topics: compression algorithms, efficient hardware-conscious implementations, compressed execution [12, 26, 19]. The goal of improving query performance using compression led to the development of lightweight compression schemes: dictionary compression, run-length encoding, frame-of-reference, delta coding, null suppression [3, 7, 16, 21, 25]. Zukowski et al. [25] proposed improved versions of these techniques that make compression more robust to outliers by separating these out as exceptions. DataBlocks [14] is a compressed storage format that reduces memory footprint through hot-cold data classification. Raman et al. [19] optimized query execution time through in-memory query processing on dictionary-compressed data in DB2 BLU.

All this work focuses on low level optimizations to either speed up query execution or improve the compression ratios. In contrast, we approach the problem of compression from a different angle. Our focus is on expressing compression as an open function to the database system, here as expression trees. Automatic learning of these expressions is based on finding patterns in data samples, as well as correlations among columns. Lee et al. [15] mentioned the possibility of exploiting columns correlations at query time. The purpose was performing join operations on columns with different encoding. In our case, we want to exploit these correlations during compression, to obtain better compression ratios.

The closest work to our research is [20]. They concatenate on correlated columns and encode them together using a variation of Huffman trees. We see this approach as heavy-weight black-box compression as it is hard-coded and relies on multiple rounds of Huffman encoding. Moreover, all the patterns in the data need to be manually supplied by the user. Our work differs in multiple ways: 1) we exploit correlated columns by sharing the same physical columns between related logical columns; 2) we apply a wide range of domain specific operators tailored to the data 3) our process is fully automated, from determining patterns and correlations between columns, to generating expressions trees.

While most compression solutions proposed so far were mainly evaluated and compared to each other on synthetic benchmarks [3, 25, 15, 14, 19], we are the first to use a comprehensive user generated dataset as the Public BI benchmark [1]. Two examples of benchmarks used for evaluating database compression and compressed execution are TPC-H [4] and its successor TPC-DS [17]. Both are synthetic, using uniform or step-wise uniform column value distributions; with fully independent columns in and between tables.

6. CONCLUSIONS

In this paper, we introduced *white-box compression*, an innovative compression model for databases. Our model automatically learns expressions from data and represents decompression in a transparent manner. It also creates opportunities for query optimization, by fusing decompression and query execution, as well as through better filter push-down and column pruning. We think white-box invites many research questions. Our next action item is to try and further compress exception columns. Another is to experiment with more adventurous column functions, such as ML models. Unexplored in this paper are also performance aspects: physical methods to quickly execute decompression expressions, rewrite techniques to make expression graphs shallower and faster; as well as more adventurous and advanced learning algorithms. Finally, the idea to make data formats contain self-descriptive compression headers raises design challenges but also systems challenges for the efficient execution of highly variate decompression sub-plans.

7. 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] P. Boncz, T. Neumann, and O. Erling. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. In *TPCTC@VLDB*, pages 61–76, 2013.

[5] A. Dziedzic, J. Wang, S. Das, B. Ding, V. R. Narasayya, and M. Syamala. Columnstore and b+ tree - are hybrid physical designs important? In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 177–190, New York, NY, USA, 2018. ACM.

[6] B. Ghiță. "Self-learning Whitebox Compression". Master's thesis, "Centrum Wiskunde & Informatica(CWI)", www.cwi.nl/~boncz/msc/2019-BogdanGhita.pdf, 2019.

[7] J. Goldstein, R. Ramakrishnan, and U. Shaft. Compressing Relations and Indexes. In *ICDE*, pages 370–379, 1998.

[8] G. Graefe and L. D. Shapiro. Data Compression and Database Performance. In *SAC*, pages 22–27, 1991.

[9] S. Héman. *"Updating Compressed Column-Stores"*. PhD thesis, Centrum Wiskunde & Informatica(CWI), 2015.

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

[11] S. Idreos, K. Zoumpatianos, B. Hentschel, M. S. Kester, and D. Guo. The Data Calculator: Data Structure Design and Cost Synthesis from First

Principles and Learned Cost Models. In *SIGMOD*, pages 535–550, 2018.

[12] A. Kemper and T. Neumann. HyPer: A Hybrid OLTP & OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *ICDE*, pages 195–206, 2011.

[13] T. Kraska, M. Alizadeh, A. Beutel, E. H. Chi, J. Ding, A. Kristo, G. Leclerc, S. Madden, H. Mao, and V. Nathan. Sagedb: A learned database system. In *CIDR*, 2019.

[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] J.-G. Lee, G. Attaluri, R. Barber, S. Idreos, M.-S. Kim, S. Lightstone, G. Lohman, et al. Joins on Encoded and Partitioned Data. *PVLDB*, pages 1355–1366, 2014.

[16] D. Lemire and L. Boytsov. Decoding Billions of Integers Per Second Through Vectorization. *Softw. Pract. Exper.*, pages 1–29, 2015.

[17] R. O. Nambiar and M. Poess. The Making of TPC-DS. In *PVLDB*, pages 1049–1058, 2006.

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

[19] V. Raman, G. Attaluri, R. Barber, G. M. Lohman, et al. DB2 with BLU Acceleration: So Much More Than Just a Column Store. *PVLDB*, pages 1080–1091, 2013.

[20] V. Raman and G. Swart. How to Wring a Table Dry: Entropy Compression of Relations and Querying of Compressed Relations. In *PVLDB*, pages 858–869, 2006.

[21] M. A. Roth and S. J. Van Horn. Database Compression. *ACM Sigmod Record*, pages 31–39, 1993.

[22] A. Vogelsgesang, M. Haubenschild, J. Finis, A. Kemper, V. Leis, T. Muehlbauer, et al. Get Real: How Benchmarks Fail to Represent the Real World. In *DBTEST*, pages 1–6, 2018.

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

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

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

[26] M. Zukowski, M. Van de Wiel, and P. Boncz. Vectorwise: A Vectorized Analytical DBMS. In *ICDE*, pages 1349–1350, 2012.