Balanced Query Processing Based on Lightweight Compression of Intermediate Results

– First TAB-Report –

Patrick Damme
Database Technology Group
Technische Universität Dresden
patrick.damme@tu-dresden.de

Supervisor:	Prof. Dr.-Ing. Wolfgang Lehner
Co-supervisor:	Prof. Dr. Thorsten Strufe
RoSI Post-Doc:	PD Dr.-Ing. habil. Dirk Habich

November 29, 2018

Abstract

Modern in-memory column-stores employ lightweight data compression to tackle the growing gap between processor speed and main memory bandwidth. However, the compression of intermediate results created during query processing has not been investigated sufficiently although accessing intermediates is as expensive as accessing the base data in these systems. Therefore, we introduce our vision of a *balanced query processing based on compressed intermediates* to improve query performance. In this report, we provide an overview of the important research challenges on the way to this goal, present our contributions so far, and give an outlook on our remaining steps.

1 Introduction

With increasingly large amounts of data being collected in numerous areas ranging from science to industry, the importance of online analytical processing (OLAP) workloads increases constantly. OLAP queries typically address a small number of columns, but a high number of rows and are, thus, most efficiently processed by column-stores, which store the data elements of one column of a table subsequently in memory. The significant developments in the main memory domain in recent years have rendered it possible to keep even large datasets entirely in main memory. Consequently, modern column-stores follow a main memory-centric architecture. These systems have to face some new architectural challenges.

Firstly, they suffer from the new bottleneck between main memory and the CPU caused by the contrast between increasingly fast multi-core processors and the comparably low main memory bandwidth. To address this problem, column-stores make extensive use of data compression. The reduced data sizes achievable through compression result in lower transfer times, a better utilization of the cache hierarchy, and less TLB misses. However, classical *heavyweight* compression algorithms such as

Huffman [11] or Lempel Ziv [24] are too slow in the context of in-memory systems. Therefore, numerous *lightweight* compression algorithms such as differential coding [16, 20] and null suppression [1, 20] have been proposed, which are much faster and, thus, suitable for in-memory column-stores. Furthermore, especially for lightweight compression algorithms, many operators can directly process the compressed data without prior decompression.

Secondly, leveraging lightweight compression optimally requires a careful consideration of the context of the data to be compressed. First of all, data might be either base data or intermediate results generated during query processing. Interestingly, in main memory-centric column-stores, accessing the intermediate results is as expensive as accessing the base data, since both reside in main memory. Thus, intermediates offer a great potential for performance improvement, which can be exploited in two orthogonal ways: (1) intermediates should be avoided whenever possible [13, 17], or (2) intermediates should be represented in a way that facilitates efficient query processing.

We focus on the second approach by investigating lightweight compression of intermediates in in-memory column-stores. This direction has not been investigated sufficiently in the literature so far. Existing systems usually keep the data compressed only until an operator cannot process the compressed data directly, whereupon the data is decompressed, but not recompressed – due to the resulting computational overhead. However, using modern hardware and state-of-the-art lightweight compression algorithms, this computational overhead can be outweighed by the benefits of compressed data. Thus, our vision is a *balanced query processing based on compressed intermediates*. That is, in a query execution plan of compression-aware physical operators, every intermediate result shall be represented using a lightweight compression algorithm suitable in the context of the surrounding operators and data characteristics such that the benefits of compression outweigh its costs.

Our research is conducted in the frame of the RoSI research training group, whose goal is the facilitation of context-sensitive software. In our case, this context is characterized by the properties of an in-memory column-store database architecture on the one hand as well as the data and queries to be processed on the other hand. To achieve our vision, we take this context into account by addressing the following three aspects of the problem: In the *structural aspect* (Section 2), we focus on efficient implementations of lightweight compression algorithms and investigate their behavior in the context of different data characteristics and hardware capabilities such as SIMD extensions. Furthermore, we explore how data can be adapted to changing contexts by transforming it to another compression format efficiently. In the *operational aspect* (Section 3), we extend the considered notion of context to the operators in a query execution plan and develop physical operators for compressed data. Finally, in the *optimization aspect* (Section 4), we develop compression-aware strategies for the database query optimizer. These strategies select a suitable compression algorithm for each intermediate in the context of a given query plan. By defining a trade-off between the objectives performance and memory consumption, these strategies are able to adapt to the context of the query execution such as the system's current resource utilization and the user's preferences.

2 Structural Aspect

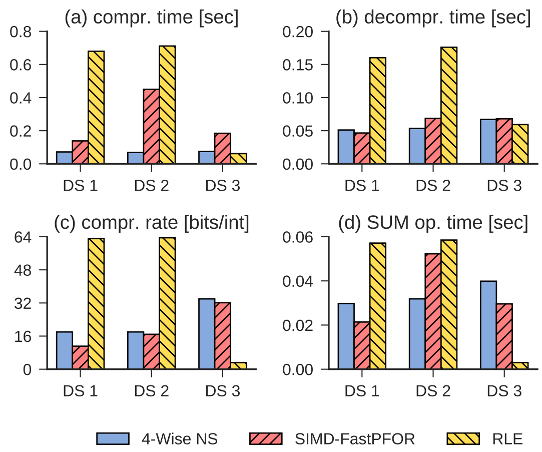
The *structural aspect* lays the foundations of this thesis by focusing on the basics of lightweight compression algorithms and on efficient transformations between the compressed formats of different algorithms. Thereby, our primary focus is on integer sequences due to their outstanding importance in column-stores: Other fixed-width data types, such as decimals, can be stored as integers and variable-width data types, such as strings, usually *need* to be represented by fixed-width integer codes from a dictionary to enable efficient processing. We have already completed our planned research in this aspect of the thesis.

2.1 Lightweight Data Compression

In the field of lossless lightweight data compression, we distinguish between *techniques*, i.e., the abstract ideas of how compression works conceptually, and *algorithms*, i.e., concrete instanciations of one *or more* techniques. So far, we consider five lightweight compression techniques for sequences of integers: frame-of-reference (FOR) [10, 25], differential coding (DELTA) [16, 20], dictionary coding (DICT) [1, 25], run-length encoding (RLE) [1, 20], and null suppression (NS) [1, 20]. FOR and DELTA represent each data element as the difference to either a certain given reference value (FOR) or to its predecessor (DELTA). DICT replaces each value by its unique key in a dictionary. The objective of these three well-known techniques is to represent the original data as a sequence of small integers, which is then suited for the actual compression using the NS technique. NS is the most studied lightweight compression technique. Its basic idea is the omission of leading zero bits in small integers. Finally, RLE tackles uninterrupted sequences of occurrences of the same value, so called *runs*. Each run is represented by its value and length. Obviously, these techniques exploit different data characteristics, such as the value range, the number of distinct values, and repeated values.

In the literature, numerous algorithms have been proposed for these techniques, e.g., [1, 10, 16, 20, 21, 23, 25], to name just a few examples. For our purposes of applying decompression and recompression *during* query execution, we depend on highly efficient implementations of these existing algorithms. One way to achieve these is to use single instruction multiple data (SIMD) extensions of modern processors, such as Intel's SSE and AVX, which allow the application of one operation to multiple data elements at once. In fact, the employment of SIMD instructions has been the major driver of the research in this field in recent years [16, 21, 23]. We have contributed to the corpus of proposed efficient implementations, e.g., through our vectorized algorithm for RLE [6], which is based on vectorized comparisons.

As lightweight compression algorithms are always tailored to certain data characteristics, their behavior in terms of performance and compression rate depends strongly on the data. Selecting the best algorithm for a given base column or intermediate requires a thorough understanding of the algorithms' behaviors subject to the data properties. Unfortunately, a sufficient comparative analysis had been missing in the literature. Thus, we conducted an experimental survey of several vectorized state-of-the-art compression algorithms from all five techniques as well as combinations thereof on numerous datasets, whereby we systematically varied the data characteristics [5, 6]. Figure 1a-c provide a sample of our results (the code was compiled using `g++ -O3` and the evaluation system was equipped with an Intel Core i7-4710MQ and 16 GB RAM). Our comparative analysis revealed several new insights. For instance, we could show how different data distributions affect the algorithms. We found that especially outliers in the distributions lead to a significant degradation in the performance and/or compression rate of certain algorithms. Furthermore, for fixed data characteristics, the best algorithm regarding performance is not necessarily the best regarding compression rate. Finally, we could show that



DS	Value distribution	Run length
1	normal($\mu = 2^{10}, \sigma = 20$)	constant(1)
2	67% normal($\mu = 2^{6}, \sigma = 20$) 33% normal($\mu = 2^{27}, \sigma = 20$)	constant(1)
3	uniform($[0, 2^{32} - 1]$)	normal($\mu = 20, \sigma = 2$)

Figure 1: Behavior of three compression algorithms (a-c) and a SUM operator on the respective compressed formats (d) for three datasets with different characteristics, each having 100M data elements.

combinations of different techniques can heavily improve the compression rate and even the (de)compression speed depending on the data. Summing up our findings, we can state that there is no single-best compression algorithm, but the choice depends on the data properties and is non-trivial. Our extensive experimental survey was made feasible by our benchmark framework for compression algorithms [7], which facilitates an efficient and organized evaluation process.

2.2 Direct Data Transformation

Assuming that the optimal compression algorithm was selected for a column, this algorithm might become sub-optimal if the data properties change *after the decision*. The properties of the base data might change over time through DML operations. While this case might be handled offline, the problem is more urgent for intermediates, whose properties can change dramatically through the application of an operator. For instance, a column containing outliers might be stored in a format that can tolerate these, perhaps at the price of a slow decompression. A selection operator might remove the outliers, making a faster non-outlier-tolerant algorithm a better choice than the original one. This motivates the need for a *transformation* of the compressed representation of the data in some *source format* to the compressed representation in some *destination format*.

A naïve approach would take two steps: (1) Apply the decompression algorithm of the source format to the data, thereby materialize the entire uncompressed data in main memory. (2) Apply the compression algorithm of the destination format to that uncompressed data. The advantage of this approach is that it builds only upon existing (de)compression algorithms. However, since it materializes the uncompressed data in main memory, it is prohibitively expensive from our point of view, since we need to transform intermediates *during* query execution.

To address this issue, we introduced *direct transformation algorithms* in [8]. This novel class of algorithms is capable of accomplishing the transformation in *one step*, i.e., without the materialization of the uncompressed data. To provide an example, we proposed a direct transformation from RLE to 4-Wise NS. In 4-Wise NS [21], the compressed data is a sequence of compressed blocks of four data elements each. The direct transformation algorithm RLE-2-4WiseNS roughly works as follows: For each pair of run value and run length in the RLE-compressed input, it creates one block of four copies of the run value, compresses it *once*, and stores it out *multiple times* until it reaches the run length. That way, it saves the intermediate store and load as well as the repeated block compression performed by the naïve approach. Our experiments showed that this and other direct transformations yield significant speed ups over the naïve approach, if the data characteristics are suitable.

3 Operational Aspect

In our currently ongoing work in the *operational aspect*, we investigate how to integrate lightweight data compression into the query execution.

3.1 Processing Model for Compressed Data

Our vision of a query processing based on compressed intermediates can best be investigated using a processing model that actually *materializes all intermediates*. Furthermore, since we focus on column-stores and since lightweight compression algorithms are designed for sequences of values, *all* intermediates should use a *columnar representation*. Hence, we chose *column-at-a-time* as the processing model.

One example of a system that uses this processing model is MonetDB [12], which internally expresses queries in the Monet Algebraic Language (MAL) [3]. The central data structure of MAL is the binary association table (BAT), which is used to represent both, base data and intermediates. Conceptually, a BAT consists of a *head* containing record ids and a *tail* containing the actual data. However, since the head always contains a dense sequence of integers, it can be omitted. Thus, a BAT is essentially just an array of data elements, making it a perfect target for lightweight compression. MAL formally defines a set of operators that consume and produce BATs, such as selection, join, and projection. We decided to use MAL as the foundation of our work, but intend to adapt MAL operators to multiple compressed formats, which we discus in the next section.

3.2 Physical Operators for Compressed Data

When adapting MAL operators to compressed data, different *degrees of integration* are possible. Figure 2 presents the cases we plan to investigate. In general, an operator might consume i inputs and produce o outputs, each of which might be represented in its individual compressed format. Figure 2a shows the baseline case of processing only uncompressed data. In the following, we assume we want to support n compressed formats for *one* operator.

A first approach to support compressed intermediates is shown in Figure 2b. The original operator for uncompressed data is surrounded by a wrapper, which temporarily decompresses the inputs and recompresses the outputs. This approach is called *transient decompression* and was proposed in [4], but to the best of our

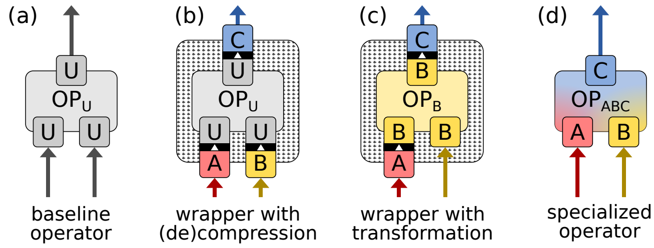


Figure 2: Integration of compression and operators. A to C are compressed formats; U is uncompressed.

knowledge, it has never been investigated in practice. For efficiency, the decompression(recompression) should not work on the entire inputs(outputs), but on small chunks fitting into the L1 cache. Changing the compressed format of the intermediates is possible by configuring the wrapper's input and output formats accordingly. The advantage of this approach is its simplicity: It reuses the existing operator and relies only on n already existing (de)compression algorithms. However, it does not exploit the benefits of working directly on compressed data.

The second approach is to adapt the operator such that it can work *directly* on compressed data (Figure 2c). Existing works such as [1, 15, 22] have already proposed *certain* operators on *certain* compressed formats. We plan to contribute to this line of research by covering the formats of recent vectorized compression algorithms. We have already investigated a SUM operator on compressed data and Figure 1d illustrates how significantly its performance depends on the data properties. We assume a common format for all inputs and outputs of the operator; for arbitrary combinations of formats, the operator is again wrapped. However, in this case the wrapper utilizes the *direct transformation* algorithms we developed in the structural aspect. Note that transformations are required only for those inputs(outputs) that are not represented in the operator's native format. The idea of bringing compressed inputs into a common format has already been proposed in [15], but only for joins on dictionary encoded data – and without *direct* transformations. We expect this approach to yield considerable speed ups compared to the first approach, since (i) the compressed data inside the wrapper is smaller, and (ii) the operator works directly on the compressed representation, such that it might, e.g., process more data elements in parallel using SIMD instructions. This approach requires n variants of the operator and $n^2 - n$ transformations, whereby the latter can be reused for all other operators. Nevertheless, the existence of a wrapper still causes a certain overhead.

The final approach tries to maximize the efficiency by tailoring the operator to a specific combination of formats (Figure 2d), making a wrapper unnecessary. Unfortunately, this approach implies the highest integration effort, requiring n^{i+o} operator variants. Thus, we intend to evaluate the potential of this approach first by considering a few promising combinations. If the results show significant improvements over the second approach, we could address the high integration effort, e.g., using code generation techniques.

The investigation of the above approaches is our current work-in-progress. Since we already have implementations of several compression and transformation algorithms available for the wrappers, our current focus is on the actual operators on compressed data. As a starting point, we decided to have a closer look and the interplay of hashing and compression, since hashing is a fundamental operation in many database operators, such as join, grouping, aggregation, and partitioning. Hash ta-

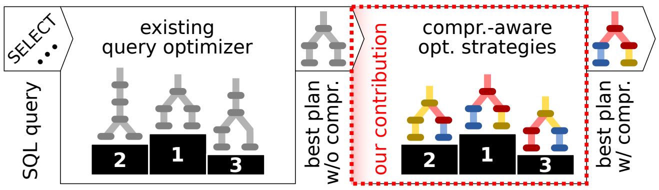


Figure 3: Compression-aware query optimization. Colors in the query plans stand for different compressed formats; grey stands for uncompressed data.

bles are a particularly important application of hashing and compression is promising here, since the reduction of the table's memory footprint could allow it to be stored closer to the CPU in the memory hierarchy. A natural way to introduce compression into a hash table would be to store the keys and values using a minimal number of bits. However, due to the random-access nature of hash tables, each key-value-pair must be easy to locate. To facilitate this in the context of compression, entries of the same bit width should be stored together. Thus, one of our ideas is the separation of a hash table into one table per key bit width. This idea could even be combined with the concise hash table (CHT) data structure proposed by Barber et al. [2], which eliminates empty buckets from a huge but sparse virtual hash table. Finally, in certain cases, it could even be possible to store tables of small compressed entries entirely in the CPU's (vector) registers.

4 Optimization Aspect

There is no single-best compression algorithm, but the decision depends on the data characteristics [6]. Moreover, given a specific dataset, one algorithm can be differently well-suited regarding different objectives, such as the compression rate or the performance. Thus, compression must be employed wisely in a query plan to make its benefits outweigh its computational overhead. This motivates the development of compression-aware query optimization strategies, our goal in the *optimization aspect*. We first elaborate on our general ideas and after that we summarize our already completed work in this direction.

4.1 Compression-aware Query Optimization Strategies

The query optimizer is one of the most complex components of a DBMS. The crucial tasks it fulfills – such as algebraic restructuring and mapping logical to physical operators – are still fundamental for compressed query execution. Due to the high complexity, deeply integrating our compression-aware strategies into an existing optimizer is beyond the scope of this thesis. Instead, we envision a second optimization phase. This phase takes the optimal plan output by an existing optimizer as input and enriches it with compression by selecting an appropriate compressed format for each intermediate and replacing the physical operators by our derived operators for compressed data (Figure 3). In the following, we briefly describe the research challenges we will have to face to achieve this goal.

Local vs. global optimization. A simple approach could be to select the best format for each intermediate in isolation. While this implies a small search space, it might fail to find the optimal plan, e.g., by changing the format too often. A global optimization, on the other hand, requires effective pruning rules to cope with the huge search space.

Creation of a cost model. Due to the complex behavior of lightweight compression algorithms and, therefore, the operators based on them, the comparison of alternative decisions should be based on a cost model. Given a set of data properties, this model must provide estimates for, e.g., the compression rate and operator runtimes. In fact, we have already developed a cost-model for (de)compression algorithms and a simple aggregation operator, which we present in the following section. As future work, we plan to extend this approach to other physical operators on compressed data.

Estimation of the data characteristics. To use the cost model effectively, the characteristics of the data must be known. However, estimating the properties of all intermediates prior to query execution is non-trivial. Erroneous estimates might result in sub-optimal decisions. Therefore, adaptive optimization strategies might be a solution.

4.2 Selecting a Suitable Compression Algorithm

The ability to select a suitable compression algorithm for a given dataset can be seen as the basis for any compression-aware query optimization strategy. Abadi et al. [1] provided a decision tree for this task. However, this tree was created manually and does not take recent algorithms into account. Other works [14, 18, 19] proposed cost-based approaches, but consider only the compression rate and not the performance.

To overcome these issues, we developed a novel cost-based selection strategy for lightweight compression algorithms [9], which is still under review. Given a set of available compression algorithms and a configuration of the data characteristics such as the number of data elements *per bit width* (a so-called *bit width histogram*), the number of distinct data elements and the sort order, our strategy selects the algorithm with the minimal *cost* with respect to a certain *objective*. Up to now, we support the the following objectives: (1) the *compression rate*, which is important for all operator variants presented in Section 3.2, since it determines the input/output size, (2) the *(de)compression performance*, which is especially important for operators with a (de)compression wrapper, and (3) the *aggregation performance*, which is a first promising step into the direction of operators directly processing compressed data. Furthermore, arbitrary trade-offs between these objectives can be defined, which allows a fine-grained adaptation to the context of the query processing. For instance, if the system is under high load and little main memory is left to the query, the compression rate could be favored over the performance; while in an interactive setting, queries might choose the algorithm with the best compression rate still meeting a certain time-constraint.

Our cost model adopts a grey-box approach. We explicitly *model* everything clear from the algorithm's specification, such as the block size the algorithm operates on. In contrast, we *measure* the impact of all hardware properties, such as the clock frequency, the cache sizes and the cost of branch mispredictions. These measurements are executed once in a calibration phase on a very small number of well-chosen datasets. For instance, in our experimental survey [6] we observed that Null Suppression algorithms mainly depend on the bit widths represented in the data. Thus, in the calibration phase, we measure the algorithms' behavior on 32 datasets, whereby in the i-th dataset all data elements have exactly i bits. We call the resulting vector of 32 measurements per algorithm and objective its *bit width profile*. We estimate the cost of a Null Suppression algorithm as the dot product of its bit width profile and the bit width histogram of the data, which could represent *any* combination of bit widths. Essentially, that way we weigh the isolated impact of each bit width by its share in the target dataset. In practice, this estimation becomes more sophisticated to address the algorithms' block sizes and their vulnerability to outliers in the data. Besides that, we also developed cost functions for logical-level algorithms as well as cascades of logical-level and physical-level algorithms.

An extensive evaluation of our cost-based selection strategy confirmed that it is able to select the actual best algorithm in most cases. In all other cases, the additional overhead incurred by a sub-optimal decision is usually small compared to the overhead of the worst-case decision. Furthermore, we could prove that our grey-box approach can automatically adapt to different hardware platforms.

5 Conclusions

Modern in-memory column-stores address the RAM-CPU-bottleneck through lightweight data compression. However, employing compression has not been investigated sufficiently for intermediate results created during query processing, although they offer great potential for performance improvement. Thus, we introduced our vision of a *balanced query processing based on compressed intermediates.* We discussed all relevant aspects of the problem in detail: (1) Our completed work in the structural aspect, where we contributed (i) an extensive experimental survey of lightweight compression algorithms in the context of different data characteristics and (ii) direct transformation algorithms facilitating the efficient adaptation of the data representation when context changes make it necessary. (2) Our ongoing work in the operational aspect, where we contribute different variants of physical operators on compressed data. (3) Our completed and future work in the optimization aspect, where we contributed a cost-based selection strategy for lightweight data compression algorithms and plan to contribute compression-aware query optimization strategies.

Acknowledgments

This work was partly funded by the German Research Foundation (DFG) within the Research Training Group "Role-based Software Infrastructures for continuous-context-sensitive Systems" (GRK 1907) and in the context of the project "Lightweight Compression Techniques for the Optimization of Complex Database Queries" (LE-1416/26-1).

References

[1] D. J. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, pages 671–682, 2006.

[2] R. Barber, G. M. Lohman, I. Pandis, V. Raman, R. Sidle, G. K. Attaluri, N. Chainani, S. Lightstone, and D. Sharpe. Memory-efficient hash joins. *PVLDB*, 8(4):353–364, 2014.

[3] P. A. Boncz and M. L. Kersten. MIL primitives for querying a fragmented world. *The VLDB Journal*, 8(2):101–119, 1999.

[4] Z. Chen, J. Gehrke, and F. Korn. Query optimization in compressed database systems. In *SIGMOD*, pages 271–282, 2001.

[5] P. Damme, D. Habich, J. Hildebrandt, and W. Lehner. Insights into the comparative evaluation of lightweight data compression algorithms. In *EDBT*, pages 562–565, 2017.

[6] P. Damme, D. Habich, J. Hildebrandt, and W. Lehner. Lightweight data compression algorithms: An experimental survey (experiments and analyses). In *EDBT*, pages 72–83, 2017.

[7] P. Damme, D. Habich, and W. Lehner. A benchmark framework for data compression techniques. In *TPCTC*, pages 77–93, 2015.

[8] P. Damme, D. Habich, and W. Lehner. Direct transformation techniques for compressed data: General approach and application scenarios. In *ADBIS*, pages 151–165, 2015.

[9] P. Damme, A. Ungethüm, J. Hildebrandt, D. Habich, and W. Lehner. From a comprehensive experimental survey to a cost-based selection strategy for lightweight data compression algorithms. *ACM Transactions on Database Systems*, Invited extension of [6], revised version currently under review, full text available upon request.

[10] J. Goldstein, R. Ramakrishnan, and U. Shaft. Compressing relations and indexes. In *ICDE*, pages 370–379, 1998.

[11] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, 1952.

[12] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten. MonetDB: Two decades of research in column-oriented database architectures. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 35(1):40–45, 2012.

[13] T. Kissinger, B. Schlegel, D. Habich, and W. Lehner. QPPT: query processing on prefix trees. In *CIDR*, 2013.

[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. Lee, G. K. Attaluri, R. Barber, N. Chainani, O. Draese, F. Ho, S. Idreos, M. Kim, S. Lightstone, G. M. Lohman, K. Morfonios, K. Murthy, I. Pandis, L. Qiao, V. Raman, V. K. Samy, R. Sidle, K. Stolze, and L. Zhang. Joins on encoded and partitioned data. *PVLDB*, 7(13):1355–1366, 2014.

[16] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Software – Practice and Experience*, 45(1):1–29, 2015.

[17] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[18] P. Przymus and K. Kaczmarski. *Transactions on Large-Scale Data- and Knowledge-Centered Systems XV: Selected Papers from ADBIS 2013 Satellite Events*, chapter Compression Planner for Time Series Database with GPU Support. 2014.

[19] A. Rasin and S. B. Zdonik. An automatic physical design tool for clustered column-stores. In *EDBT*, 2013.

[20] M. A. Roth and S. J. Van Horn. Database compression. *SIGMOD Record*, 22(3):31–39, 1993.

[21] B. Schlegel, R. Gemulla, and W. Lehner. Fast integer compression using SIMD instructions. In *DaMoN*, pages 34–40, 2010.

[22] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. SIMD-Scan: Ultra fast in-memory table scan using on-chip vector processing units. *PVLDB*, 2(1):385–394, 2009.

[23] W. X. Zhao, X. Zhang, D. Lemire, D. Shan, J. Nie, H. Yan, and J. Wen. A general simd-based approach to accelerating compression algorithms. *ACM Transactions on Information Systems*, 33(3):15:1–15:28, 2015.

[24] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[25] M. Zukowski, S. Héman, N. Nes, and P. A. Boncz. Super-scalar RAM-CPU cache compression. In *ICDE*, 2006.