MorphStore: Analytical Query Engine with a Holistic Compression-Enabled Processing Model

Patrick Damme, Annett Ungethüm, Johannes Pietrzyk, Alexander Krause,
Dirk Habich, Wolfgang Lehner
Database Systems Group
Technische Universität Dresden
Dresden, Germany
{firstname.lastname}@tu-dresden.de

ABSTRACT

In this paper, we present *MorphStore*, an open-source in-memory columnar analytical query engine with a novel *holistic compression-enabled* processing model. Basically, compression using lightweight integer compression algorithms already plays an important role in existing in-memory column-store database systems, but mainly for base data. In particular, during query processing, these systems only keep the data compressed until an operator cannot process the compressed data directly, whereupon the data is decompressed, but not recompressed. Thus, the full potential of compression during query processing is not exploited. To overcome that, we developed a novel compression-enabled processing model as presented in this paper. As we are going to show, the continuous usage of compression for all base data and all intermediates is very beneficial to reduce the overall memory footprint as well as to improve the query performance.

PVLDB Reference Format:
Patrick Damme, Annett Ungethüm, Johannes Pietrzyk, Alexander Krause, Dirk Habich, Wolfgang Lehner. MorphStore: Analytical Query Engine with a Holistic Compression-Enabled Processing Model. *PVLDB*, 13(11): 2396–2410, 2020.
DOI: https://doi.org/10.14778/3407790.3407833

1. INTRODUCTION

With increasingly large amounts of data being collected in numerous application areas ranging from science to industry, the importance of online analytical processing (OLAP) workloads increases constantly [15]. OLAP queries typically access a small number of columns, but a high number of rows and are, thus, most efficiently processed by column-stores [15]. Moreover, the significant developments in the main-memory domain have rendered it possible to keep even large data sets entirely in main-memory. For these reasons, in-memory column-stores have established themselves as state-of-the-art database management systems (DBMS) for OLAP workloads [13, 25, 72].

Proceedings of the VLDB Endowment, Vol. 13, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3407790.3407833

In these systems, all values of every column are encoded as a sequence of integer values and, thus, query processing is done on these integer sequences [1, 2, 25]. To improve query processing performance in this case, vectorization based on the *Single Instruction Multiple Data (SIMD)* parallel paradigm is a state-of-the-art technique [27, 39, 58, 61, 74, 83], because all mainstream CPUs offer powerful SIMD extensions nowadays [36, 74]. The main goal of SIMD is to increase the single-thread performance by executing an identical operation on multiple data elements in a vector register simultaneously (data parallelism) [36]. Aside from vectorization, lightweight integer compression algorithms also play an important role [1, 2, 25, 45]. With the help of some additional lightweight computations for integer compression, the necessary memory space can be reduced [1, 2, 25, 45]. As we have shown in [18, 20], there is a large variety of lightweight integer compression schemes available and there is no single-best algorithm, but the decision depends on several factors, most importantly on the data characteristics. Moreover, compressed integer values also offer advantages for data processing such as increasing the effective bandwidth to reduce the memory wall, a better utilization of the cache hierarchy, and more logical data elements within a physical vector register. Nevertheless, (de)compression leads to additional computational effort, which is typically kept low by means of vectorization [18, 20, 48, 57, 82].

Thus, vectorization and integer compression complement each other in a suitable way and the smart use of both techniques can improve the single-thread query performance. This is especially true when a direct processing of compressed integer values is possible [26, 27, 45, 46, 51, 77]. For example, several column scan approaches have been presented in the literature, where filter predicates are directly evaluated on compressed integer data [27, 51, 77]. However, existing systems only provide a very limited set of compression algorithms for base columns [1, 2, 25, 45]. Furthermore, during query processing, these systems only keep the data compressed until an operator cannot process the compressed data directly, whereupon the data is decompressed, but not recompressed [1, 2, 25, 45]. Thus, the potential of compression during query processing is not fully exploited.

Core Contribution. To enable this potential as advertised in Figure 1, we designed a novel *holistic compression-enabled processing model* satisfying four design principles: (**DP1**) Our model is a compression-enabled optimization of the well-known operator-at-a-time processing model introduced by MonetDB [13, 37]. Thus, all intermediate results

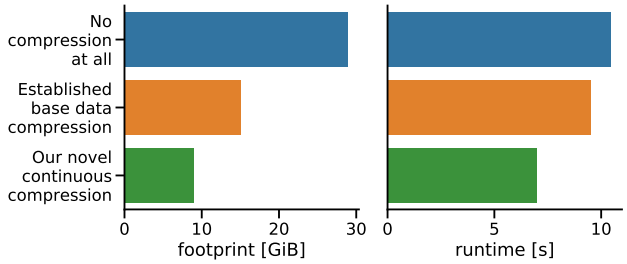

Figure 1: Advertising the benefits of compression in a vectorized columnar query processing for the SSB [66] at SF 100. Averages of all 13 queries. A detailed description can be found in Section 5.

should be representable using a lightweight integer compression algorithm. With that, we want to enable the *continuous* usage of compression for the whole query execution from base to intermediate data. (**DP2**) Since data characteristics have an impact on the compression scheme decision and these usually change during query processing, a suitable scheme should be chosen for each intermediate from a rich and easily extensible set of schemes. Moreover, the selection for each intermediate should not depend on the scheme used for another one to independently adapt to its particular data characteristics. This implies that a change of the compression scheme from one intermediate to the next should be possible in a very efficient way. (**DP3**) No physical columnar query operator should require the uncompressed materialization of its entire input or output data, since this would severely limit the benefits achievable through compression. In particular, a full decompression of the input data should be avoided. (**DP4**) Related work in this domain is manifold as described in Section 2 and our approach should build on as well as extend this work in a seamless and efficient way.

To prove the benefits of our approach, we developed the open-source analytical query engine *MorphStore* [32], which supports vectorized processing using Intel's latest SIMD extension AVX-512. As depicted in Figure 1, going from a purely uncompressed processing to a state-of-the-art processing with compressed base data and uncompressed intermediates reduces the average memory footprint of all 13 SSB queries at scale factor 100 by 48%, while the query runtimes can be reduced by only 9%. However, our novel approach unlocks significant further improvements. In particular, the memory footprints can be reduced by *additional* 21%, and the runtimes by *additional* 24%. Thus, the continuous compression of intermediates contributes decisively to the reduction of memory footprints and runtimes.

Contributions in Detail and Outline. To present *MorphStore* in detail, we make the following contributions:

1. We give a comprehensive and systematical overview of preliminary work in Section 2.
2. Based on that, we describe our *holistic compression-enabled processing model* in Section 3. In particular, we introduce a novel *morphing-wrapper* for operators.
3. In Section 4, we introduce *MorphStore* as an implementation for our novel processing model.
4. Then, we present selected results of our exhaustive evaluation using micro-benchmarks and the SSB [66] in Section 5. We show that the continuous use of compression is very beneficial from two perspectives: *memory footprint* and *query runtime*.
5. Our evaluation shows that the choice of the compression scheme for intermediates opens up a new dimension for query optimization. Thus, we highlight and present some initial steps for a compression-aware optimization in Section 6.

Finally, we discuss our approach in a broader context in Section 7 and briefly summarize the paper in Section 8.

2. PRELIMINARIES

Without claim of completeness, we review related work on vectorization and lightweight integer compression for in-memory column-store systems in this section.

2.1 Vectorization

Vectorization based on the SIMD parallel paradigm is typically applied to isolated query operators [2, 58, 83]. Many vectorized implementations for joins [5, 6, 9] and sorting [17, 59, 67] have been proposed. Moreover, linear access operators such as column scans are well-investigated [27, 77]. Non-linear access operators, such as hash-tables and partitioning have also been investigated and comprehensively evaluated [56, 58, 60]. In addition to that, vectorization and prefetching have been combined to minimize cache misses on the query level [52]. However, vectorization only increases the performance in terms of calculations by executing the same operation on multiple data elements simultaneously [36]. This further increases the gap between computing power of CPUs and main-memory bandwidth (memory wall). Thus, vectorization can reduce query runtimes to some degree, but the achieved reduction is suboptimal because query operators quickly become memory-bound.

2.2 Lightweight Integer Compression

To efficiently address this memory wall effect, compression is a state-of-the-art technique in in-memory column-stores.

2.2.1 Compression Algorithms

Since in-memory column-store systems encode and process all values as integers [1, 2, 25], these systems extensively use lightweight lossless integer compression [1, 2, 18, 25]. A large corpus of such integer compression schemes has been developed, and we have to distinguish between *techniques*, *algorithms*, and *implementations* [18, 20, 34, 48].

Techniques: Five basic techniques are known and frequently used: frame-of-reference (FOR) [28, 86], delta coding (DELTA) [48, 65], dictionary encoding (DICT) [2, 8, 65, 86], run-length encoding (RLE) [2, 65], and null suppression (NS) [2, 65]. FOR and DELTA represent each value as the difference to a certain given reference value or to its predecessor value, respectively. DICT replaces each value by its unique key given by a dictionary. The objective of these three techniques is to represent the original data as a sequence of small integers, which is then suited for actual compression using NS. NS is the most well-studied technique and its basic idea is the omission of leading zeros in the bit representation. Finally, RLE tackles uninterrupted sequences of occurrences of the same value, so-called runs. In its compressed format, each run is represented by its value and run length. Thus, the compressed data is a sequence of such pairs. To sum up, FOR, DELTA, DICT, and RLE address the logical level, while NS considers the physical level.

Algorithms: The genericity of these techniques is the foundation to tailor lightweight integer compression algorithms to different data characteristics [18, 20]. A concrete algorithm can be described as a cascade of one or more of these basic techniques. On the algorithm level, NS has been studied most extensively [3, 4, 22, 24, 48, 50, 57, 64, 68, 69, 71, 79, 81, 82, 86]. An example NS algorithm is *Binary Packing (BP)* [48, 69]. The basic idea of BP is to partition a sequence of integer values into blocks and compress every value in a block using a fixed bit width (namely, the effective bit width of the largest value in the block). Based on that, BP can adapt the bit width per block. The logical-level techniques, however, have not received much attention *on the algorithm level*. Different algorithms for RLE differ, e.g., in the way they record the repetitions. The run value can be stored together with the run length [49], or the run's start position in the input sequence [49], or both [2]. Furthermore, runs of length one, which allow no actual reduction of the number of data elements, could be stored in a special way to avoid the overhead of storing the run length [49]. DELTA, FOR, and DICT have usually been investigated in connection with the NS technique [22, 86].

Implementation: An *implementation* is a hardware-specific executable code. In recent years, the efficient *vectorized* implementation to reduce the computational overhead has attracted most of the attention [48, 57, 68, 71, 73, 82]. The focus of those vectorized implementations is first and foremost on NS algorithms. For example, SIMD-BP [48] is a vectorized implementation of the NS algorithm BP. In comparison, the logical-level techniques have been neglected, although there are also some papers presenting vectorized implementation for DELTA [48] or RLE [20, 73].

Experimental Survey. In [18, 20], we presented an extensive experimental survey of this large corpus of lightweight integer compression algorithms and implementations. By analyzing the evaluation metrics compression rate, compression speed, decompression speed, and aggregation speed, we could conclude that there is no single-best lightweight compression algorithm, but the choice is non-trivial and depends on: (i) the data characteristics, (ii) the hardware properties, (iii) the SIMD extension, and (iv) the objective. The latter point means that the best algorithm with respect to the compression rate is not necessarily optimal regarding (de)compression speed. This allows interesting trade-offs between these two fundamental objectives for optimization.

2.2.2 Query Operators on Compressed Data

A major advantage of lightweight integer compression is that some query operators can process the compressed data directly, without decompression. For that, there is a lot of related work available and we can classify them into processing data compressed at the logical and the physical level.

Logical Level: DICT was the first compression scheme whose qualification for direct processing was investigated. In [30], the authors identified that the equality-preserving property of DICT can be exploited by query operators employing exact-match comparisons. For instance, selection scans can process dictionary keys directly, without looking up the actual values in the dictionary, by mapping the selection predicate's constants to keys using the same dictionary as for the data. Furthermore, they note that duplicate elimination, grouping, equality joins, and set operations can directly work on dictionary keys. Especially for equality joins and set operations, the authors explicitly assume the same dictionary across different columns of the same domain.

In contrast to that, Lee et al. [46] proposed to encode each attribute individually to exploit skew in the data distribution for an efficient encoding. However, this implies that join-operators can no longer compare dictionary keys from different inputs directly any more. To address this issue for hash joins, they propose to perform a so-called *encoding translation* by re-encoding the keys of the build-side with the dictionary of the probe-side. After the dictionary encodings have been reconciled, dictionary keys, i.e., compressed values are inserted into the hash table to avoid decompression in both the build phase and the probe phase.

Additionally, Abadi et al. [2] sketched how database operators can directly process run-length encoded data. In detail, they mention that an RLE-compressed inner of a nested loop join does not need to be decompressed. Instead, the data elements in the outer need to be compared only to the run values of the inner. Each time a comparison succeeds, multiple join matches are found at once, whereby the number of matches is the corresponding run length in the inner. Furthermore, a summation on a run-length encoded column can be done by summing up the products of corresponding pairs of run value and run length.

Physical Level: Most DBMSs support different data types for integral attributes and offer physical query operators tailored to these types. For instance, the SQL standard defines the types `TINYINT`, `SMALLINT`, `INTEGER`, and `BIGINT`, which represent a single value using 1, 2, 4, and 8 bytes, respectively. Representing data in one of these types is, perhaps, the simplest form of lightweight compression. Thus, it can be seen as a particular NS format, namely a form of BP, whereby a common bit width is used for all elements of a column and this bit width must be either 8, 16, 32, or 64, making the compressed data byte-aligned. Thus, most DBMSs can implicitly process compressed data directly.

Even recent research on compressed processing frequently restricts the physical-level compression to byte-aligned NS [2, 7, 45, 76]. This has two reasons: (i) byte-alignment suits the byte-addressability of main-memory naturally and (ii) data elements of 8, 16, 32, and 64 bits can be processed natively on current microprocessors using both classical scalar and modern vectorized instructions. The second reason is especially important, since it implies that all operators can be tailored to directly work on compressed data with a low effort. However, some authors observed that such simple byte-aligned packing approaches lack support for arbitrary bit widths and work at a sub-optimal bit-level parallelism [27, 51, 77]. Thus, some query operators have been proposed for more sophisticated physical-level compression formats.

In this direction, the column scan has received a lot of attention. For example, Willhalm et al. proposed *SIMD-Scan*, a full column scan algorithm for data packed with arbitrary bit widths [77]. They observed that such full column scans are usually memory-bound, but become compute-bound when performed on compressed data. To alleviate this, they focused on a vectorized implementation using Intel's SSE instructions on 128-bit vector registers. In [51] Li and Patel criticize that the *SIMD-Scan* algorithm suffers from a sub-optimal bit-level parallelism, since the employed 32-bit comparisons effectively waste available bits, if the bit width is below 32. To address this issue, they propose so-called *bit-parallel methods*. In particular, the authors intro-

duce two pairs of a physical memory layout and a column scan algorithm efficiently processing data in this layout using an algebraic framework. Feng et al. [27] build upon this work and especially focus on SIMD instructions. Their *ByteSlice* approach comprises a new memory layout and a suitable scan algorithm. Besides selection, aggregation has been studied for data compressed at the physical level, e.g., Feng and Lo proposed bit-parallel implementations of common aggregation functions directly on compressed data [26].

2.2.3 Compression and Query Execution

Besides individual operators for compressed data, there are different ways to employ compression in a query as a whole. Chen et al. [16] intensively investigated the integration of compression into query execution in the context of disk-centric row-stores and introduced three strategies: (i) **Eager decompression** loads data into main-memory, immediately decompresses it, and the entire query processing takes place on uncompressed data. (ii) **Lazy decompression** keeps data in compressed form during query processing as long as possible. Base data and, perhaps, early intermediate results are represented in a compressed way and processed by specific operators. However, as soon as an operator cannot process the compressed data directly, the data is decompressed and from this point, all processing happens entirely on uncompressed data, which incurs unnecessarily large intermediates and wastes the potential of working on compressed data directly. (iii) **Transient decompression** temporarily decompresses the inputs for operators incapable of processing compressed data directly and *keeps the compressed input elements*. Thus, the processing is done on uncompressed data elements, but for the output, the compressed input elements are used again, such that subsequent query operators can still benefit from compression.

For in-memory column-store systems, the *lazy decompression* strategy has been investigated in several works [21, 45, 63, 86]. For example, [45] proposes a storage format for cold (analyical) data subdividing a column into blocks, each of which can be represented in its individual compressed format. The formats to choose from are a variant of RLE as well as FOR and DICT combined with a byte-aligned NS algorithm. Based on that, they present an approach to integrate a vectorized column scan into the compilation-based query engine of HyPer [38]. In particular, their scan outputs a list of logical positions in the compressed column, which are subsequently extracted and decompressed. After that, the query execution continues with uncompressed data. Another example is Oracle's in-memory engine [21] executing only selection scans on compressed data as well.

A more sophisticated approach was presented in [2]. The authors show how they integrate five compressed formats into the query execution of the column-store *C-Store* [72]. They observe that supporting n compressed formats requires n variants of all unary query operators and n^2 variants of all binary query operators. Since this amounts to a high total number, their main focus is on the reduction of the integration effort. Most importantly, each base and intermediate column is split into blocks that can be accessed by operators via a special API. This API abstracts from the particular format by exposing properties exploitable by query operators, e.g., whether the data in the block is sorted or whether the block contains only one distinct value. Consequently, query operators are not specialized to individual formats, but to such properties that can reduce the number of variants. As a fallback, the API allows to decompress the block so that the operator can iterate over the uncompressed data elements. The authors' extension to C-Store also supports compressed intermediates. However, an operator's output format is hard-coded for each (combination of) input format(s) and chosen depending on what seemed to the authors easy to implement. In particular, the data characteristics of the intermediates are not taken into account.

In Hyrise [10, 11, 23], base data is represented as columns horizontally partitioned into segments, whereby each segment can have its individual compressed format. To limit the effort of integrating compression into the query execution, the authors also decide to introduce a layer of abstraction. In particular, they implement iterator-based methods for sequential and random access to each compressed format. Query operators make use of the general iterator-interface irrespective of the underlying compressed formats of their inputs. Intermediate results are not explicitly compressed.

The only work we are aware of that explicitly investigates the compression of intermediate results is [31]. Unfortunately, the authors focus only on complex queries over *bit vectors*, i.e., another data structure, with the operators `AND`, `OR`, and `XOR`. Furthermore, they distinguish only between uncompressed and compressed data, but not between different compressed formats. Nevertheless, their motivation is similar to ours: They also observe that the characteristics of intermediate bit vectors may change during query processing, rendering either the compressed or the uncompressed representation more suitable on a *per-intermediate* basis. Therefore, they want to support operators on all possible combinations of (un)compressed inputs and outputs. While they can reuse (un)compressed-only operators from previous works, they contribute variants for mixed compressed and uncompressed inputs. Moreover, they decouple the output format from the input formats by reusing existing methods to append to (un)compressed bit vectors.

2.3 Lessons Learned

To improve query performance, vectorization and integer compression are state-of-the-art in-memory column-stores. The approaches to integrate compression into query operators range from a generic and transparent decompression during the reading data access [86] over abstractions for compressed formats [2] to highly format-specific algorithms for particular operators [26, 27, 51, 77]. Thus, an operator executes its operations either on the uncompressed format of data elements (*uncompressed operator*) or on data elements *compressed* according to a particular compression format for each of its input and output columns (*specialized operator*), as illustrated on the left side in Figure 2. However, the main focus of compression was so far on the storage and processing of *base columns*, which decreases the query runtime compared to a vectorized processing of uncompressed data as highlighted in Figure 1. To the best of our knowledge, a *systematic* investigation of the *continuous* use of lightweight integer compression for intermediate results in in-memory column-stores has never been addressed before.

3. COMPRESSION-ENABLED MODEL

The overall goal of our *holistic compression-enabled processing model* is to enable the continuous use of lightweight integer compression for intermediate results while pursuing

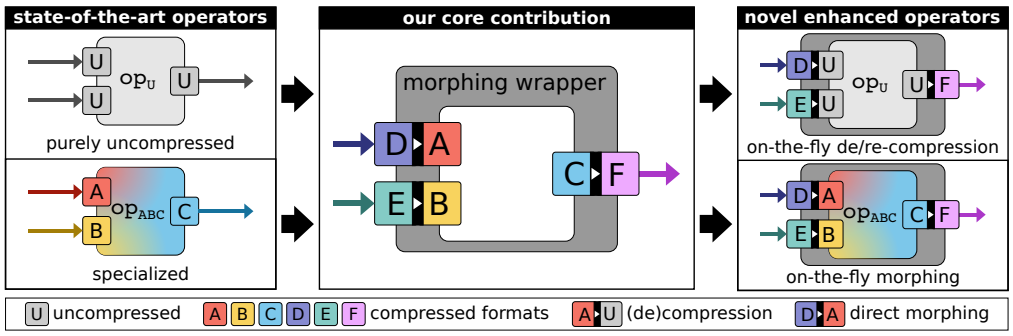

Figure 2: From state-of-the-art operators to novel enhanced operators using our proposed *morphing-wrapper* at the example of an arbitrary operator op with two inputs and one output.

the four design principles introduced in Section 1. Thus, the major challenge is the interplay between query operators and the large corpus of compression formats. A possible solution would be the development of a large set of *specialized operators*. However, the most decisive drawback of *specialized operators* is the high conceptual and integration effort they incur. To support all combinations of n compressed formats for the i inputs and o outputs of one operator, n^{i+o} variants of that operator would be required. Unless all of these variants are available, the choice of the intermediates' formats is restricted, since they must match those expected by the operators, thereby impeding design principle DP2.

3.1 Morphing-Wrapper for Operators

To achieve the highest flexibility for the choice of the intermediates' formats, we propose a novel *morphing-wrapper* concept as illustrated in Figure 2. Our concept is inspired by *transient decompression* [16], but we enhance this approach in several directions to satisfy our design principles.

As shown in Figure 2, we propose surrounding a query operator with a wrapper decoupling the operator's input and output formats from the formats used to materialize the columns in memory. Our *morphing-wrapper* can be applied to uncompressed as well as specialized operators leading to two new enhanced operator classes: *on-the-fly de/re-compression operators* and *on-the-fly morphing operators*. Thus, we are able (i) to execute each operator on every (un)compressed input format and (ii) to output the operator result in an arbitrary (un)compressed format. For that, the task of our *morphing-wrapper* is to morph the input data from the wrapper's input format to the operator's input format as well as to morph the output data from the operator's output format to the wrapper's output format. Note that our *morphing-wrapper* allows arbitrary changes of the compressed format from the input to the output of an operator during the query execution by configuring the wrapper accordingly, which is impossible with transient decompression.

Thus, we clearly separate the data representation using lightweight compression formats from the query operators, but also define a clear interface for cooperation. The applicability and efficiency of our morphing-wrapper depends on the *granularity* of the data exchange between wrapper and operator. At the coarsest granularity, the wrapper would morph an **entire column** before passing it to the operator, respectively receive an entire column from the operator for the morphing. That is, an uncompressed column might be created within the wrapper, thereby contradicting DP3.

A more reasonable granularity is the **Lx-cache-resident block**. The basic idea is not to materialize morphed data in main-memory, but only in the cache hierarchy. On the input side, the wrapper morphs a block of column values when required and pipelines only this morphed block to the operator. On the output side, the wrapper first buffers the operator output, and morphs it to the appropriate format when the block size is reached. Hence, morphing and operator execution are separated at the function-/routine-level. The block size should be chosen in a cache-conscious way to guarantee that the morphed data is still in the Lx-cache when the processing starts and the output data is still in the Lx-cache when the wrapper issues the morphing of an output block. Note that this approach is related to RAM-cache decompression [86] with the difference that our morphing is designed to also output compressed data, thereby allowing for compressed intermediates.

The finest granularity possible with a state-of-the-art vectorized processing is the **vector register**. On the input side, the wrapper decompresses a single vector at a time and pipelines it to the operator. On the output side, the operator produces vectors of data elements, which it pipelines to the wrapper for instantaneous morphing. Thus, the borders between morphing and operator execution are blurred, as these are separated only at the instruction-level. Unfortunately, this granularity is not possible for all combinations of *output* formats and operators. Some lightweight compression algorithms need to analyze a certain number of data elements to decide how to compress them. For instance, SIMD-BP128 [48] needs to determine the maximum bit width of a block of 128 data elements before it can pack the data. This bit width cannot be known when just one vector register of uncompressed data elements is available at a time.

3.2 Novel Enhanced Operators

In *on-the-fly de/re-compression* operators, our *morphing-wrapper* conducts a decompression on the input side and compression on the output side. Although uncompressed data is processed internally, the performance of these operators can profit from the reduced memory footprint of their input and output columns, which leads to an increased effective memory bandwidth. To support n compressed formats for one operator, the original operator for uncompressed data is reused. Additionally, only n compression and n decompression algorithms are required, which can be used for wrapping other operators as well. This is currently the most important enhanced operator class for us, since it can eas-

ily be implemented, but already fulfils our design principles DP1 to DP3. Nevertheless, it does not exploit the potential of working directly on compressed data.

In contrast to that, in *on-the-fly morphing operators*, our morphing-wrapper employs so-called *direct morphing algorithms* capable of changing the data representation from one compressed format to another one [19]. Thus, these operators can process compressed data internally by wrapping a specialized operator tailored to certain formats, while they can handle inputs and outputs in *different* formats. In addition to the footprint reduction and effective bandwidth increase mentioned above, on-the-fly morphing also allows the direct processing of compressed data, which can be more efficient due to a higher data-level parallelism. Supporting n compressed formats for one operator requires $n^2 - n$ direct morphing algorithms, which can be reused for other operators. The idea of adapting the compressed formats of an operator's inputs has already been sketched in [46]. However, their *encoding translation* addresses only (i) the inputs, (ii) join-operators, and (iii) DICT-compressed data. In contrast, our proposed *on-the-fly morphing* (i) can also be applied to an operator's outputs, (ii) is possible for any query operator, and (iii) supports arbitrarily compressed formats. Finally, *on-the-fly morphing* unifies the virtues of *on-the-fly de/re-compression* and specialized operators.

3.3 Query Execution

A query execution plan (QEP) exploiting continuous compression is constructed using our *novel on-the-fly de/re-compression* and *on-the-fly morphing operators* in the same manner as for state-of-the-art uncompressed or specialized operators since the operators' inputs and outputs still have the same semantics and only differ in the compressed formats. In such a QEP, each base column and intermediate result has a particular (un)compressed format, which can be chosen flexibly. To ensure that the plan is *consistent*, the expected formats of each operator must match those of the accessed input and output columns. Our morphing wrapper can be used to ensure this. Furthermore, the final query result columns should always be uncompressed, since they are returned to the client application. Finally, it is also possible to mix state-of-the-art operators with our novel enhanced operators in one QEP. The choice of the class for a particular operator depends on: (i) the availability of the respective physical operator, and (ii) typical objectives of query optimization, such as memory footprint or query runtime.

4. MORPHSTORE ENGINE

Our open-source analytical query engine *MorphStore*[1] implements our *holistic compression-enabled* processing model in C++ [32, 33]. All query operators, (de)compression, and morphing algorithms are vectorized through our recently introduced *Template Vector Library* (TVL) [74], which allows to write vectorized C++ code independently of any particular SIMD extension. Thus, in the following, we abstract from a particular SIMD extension and vector length.

4.1 Columnar Storage

Base columns, intermediates, and query results are of exactly the same nature, whereby the elements of each column are unsigned integers. If the integers for a base column were obtained through a dictionary coding, we assume an individual dictionary per domain as proposed in [46]. If range predicates need to be evaluated, we assume the dictionary coding to be order-preserving. Otherwise, the equality-preserving property of DICT suffices for point predicates.

We decided to focus on (unsigned) *64-bit* integers as the data type of the uncompressed data, since this is the native word width of most common microprocessors nowadays. As most of the literature on lightweight integer compression [48, 68, 71, 82] assumes 32-bit integers, we reimplemented some algorithms for 64-bit data elements. In particular, at the logical level, we currently support (i) DELTA and (ii) FOR, and at the physical level, we currently support the NS schemes (i) Static BP (a variant of BP with one block and a fixed bit width for all data elements) and (ii) SIMD-BP [48] (see Section 2.2.1)[2]. Additionally, cascades of one logical-level and one physical-level algorithm are possible.

MorphStore's column data structure is a continuous buffer of bytes. Therein, the entire data of the column is stored either uncompressed or compressed according to *exactly one* of the formats mentioned above[3]. Some lightweight integer compression algorithms are able to compress integer sequences of arbitrary lengths. However, others subdivide the data into blocks of a certain number of data elements and cannot represent smaller amounts of data, e.g., SIMD-BP512, our port of SIMD-BP128 to AVX-512 [20], assumes blocks of 512 data elements each. To be able to deal with columns of arbitrary lengths, each column is subdivided into a compressed main part and an uncompressed remainder. Assuming a column of n data elements and a compression algorithm with a block size of bs, the compressed part contains the first $\lfloor n/bs \rfloor$ data elements of the column represented in the column's compressed format and the remainder contains the remaining $n \mod bs$ data elements as uncompressed 64-bit integers. The remainder is stored directly behind the compressed data in the column's buffer and has to be taken into account by operators, too. A separate structure of meta data stores the sizes of these two parts.

4.2 Wrapper and Query Operators

Our query operators are strongly inspired by those of MonetDB [12][4], whereby our current implementation focuses on a set of query operators that are sufficient to execute the Star Schema Benchmark (SSB) [66]. Nevertheless, since our *morphing-wrapper* is not operator-specific, other query operators could be embraced in the same way. We started with the implementation for a purely uncompressed processing, which involves no compressed data at all and serves merely as a baseline.

Then, we focused on *on-the-fly de/re-compression* to realize our *morphing-wrapper* in combination with the available uncompressed operators. This already enables a continuous use of compression for all intermediates (and base columns)

[1] `https://morphstore.github.io`

[2] Integrating more algorithms is straightforward with our morphing-wrapper, and we plan to extend the choice.

[3] This is not a drawback, since most lightweight integer compression algorithms, e.g., DELTA, FOR, and SIMD-BP, as well as the cascades mentioned above, adapt to local variations in the data distributions internally *on their own*.

[4] Initially, MonetDB's operators processed BATs consisting of a head and a tail column [13, 37], but meanwhile, these operators have been re-engineered to work on headless BATs, i.e., mere sequences of values [70].

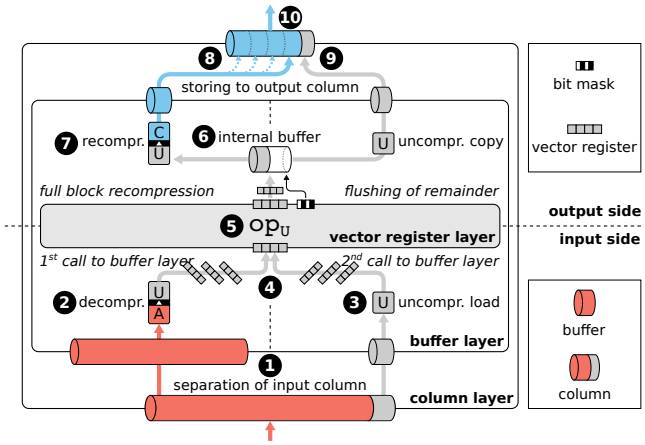

Figure 3: Execution of a query operator adopting on-the-fly de/re-compression.

with a low integration effort. Generally, a naïve implementation of *on-the-fly de/re-compression* operators would suffer from either (i) low performance due to, e.g., virtual function calls, or (ii) high source code duplication due to the explicit implementation of all combinations, resulting in hardly maintainable source code. To avoid both of these issues, we apply a number of specific techniques.

On the input side, our *morphing-wrapper* works on a vector register-granularity to avoid the materialization of uncompressed data even in caches. However, due to the reasons mentioned in Section 3, this is not always possible *on the output side*. Thus, we use the granularity of an Lx-cache-resident block here.

As shown in Figure 3, each *on-the-fly de/re-compression* operator follows a division-of-concerns approach by employing *three layers* with the following responsibilities: The *column layer* provides the interface to the operator and takes care of the subdivision of the column into a compressed main part and an uncompressed remainder. The *buffer layer* realizes the *morphing-wrapper*. The *vector register layer* represents the actual *operator core* working on uncompressed data. The name of each layer indicates the unit of data it consumes and produces. Each layer is implemented as a C++ function, whereby the column and buffer layers require individual functions for the *input side* and the *output side*.

In the following, we describe the execution of an operator with one input and one output column, both of which are accessed *sequentially*. With the numbers in brackets, we refer to those in Figure 3. The column layer identifies the compressed main part and uncompressed remainder of the input column with the help of the column's meta data. Then, it calls the input-side buffer layer once for each of these two sub-buffers (1). The input-side buffer layer is essentially the decompression routine of the input column's compressed format (2) or a simple loading of uncompressed data for the remainder (3). However, instead of storing decompressed vectors to memory, it passes each of them to the operator core at the vector register layer (4). The vector register layer consumes vectors of uncompressed 64-bit data elements. It executes the respective operator on each vector, whereby a vector of uncompressed output data elements is produced (5). For selective operators, a bit mask indicates which of the output vector's elements are valid. This output vector, perhaps accompanied by a bit mask, is handed over to the output-side buffer layer. The output-side buffer layer accepts one uncompressed vector at a time and appends it to its uncompressed internal Lx-cache-resident buffer, whereby the valid data elements indicated by the bit mask are stored compactly (6). Once this internal buffer reaches its capacity, the compression algorithm of the output column's format is called (7). It loads uncompressed data from the internal buffer and appends compressed data to the output column's buffer (8). After both calls to the input side buffer layer have terminated, the possible remaining data elements in the output side buffer layer's internal buffer need to be flushed. Of this remaining data, as much as possible is appended to the output column in compressed form, while the possible remainder is appended uncompressed (9). Finally, the output-side column layer returns the output column to the caller of the operator (10). Note that the input side and output side work in an *interleaved* fashion, i.e., the output side is active before the input side returns.

Implementing our morphing-wrapper, the buffer layer is the only layer depending on the particular format, while the column layer only needs to know the format to invoke the buffer layer correctly, and the vector register layer is not concerned with formats at all. For the same reason, the buffer layer is the only layer that is *not specific to the logical operator*, while the vector register layer obviously is, and the column layer needs to know the operator core to pass to the buffer layer. This constellation enables an economical, non-repetitive implementation based on C++ template metaprogramming. In particular, the formats of the input and output columns are modeled as template parameters of the column and buffer layers. The column layer is implemented generically with respect to these formats. However, for both the input side and the output side of the buffer layer, template specializations must be provided for each format to be supported. These specializations are strongly based on the decompression and compression algorithms of the formats, respectively. Thus, these specializations can be reused by all operators. Furthermore, the input-side buffer layer receives the core operator to call as a template parameter as well. Finally, the column layer has to initialize the vector register layer such that it calls the output format's template specialization of the output-side buffer layer. The use of templates prevents expensive *virtual* function calls, since the right specializations are determined at compile-time. Besides that, expensive frequent calls to the vector register layer are avoided by forcing the compiler to inline it into the input-side buffer layer.

Almost all our physical operators access both their input and output data sequentially. However, there are some operators employing random read or write access. For example, the `project`-operator requires random read access to compressed data, because this operator is used, for instance, to transfer the result of a selection on one column (sequential access) to another column (random access). Since lightweight integer compression algorithms are designed for efficient sequential access, random access incurs some challenges. At the logical level of compression, the interpretation of one particular compressed data element might require either meta information, as for FOR and DICT, or even information on all preceding data elements, as for DELTA. At the physical level of compression, the challenge is three-

Table 1: Properties of our synthetic columns.

	Data distribution	Sorted	Maximum bit width
C0	uniform in $[0, 7]$	no	3
C1	uniform in $[0, 63]$	no	6
C2	99.99% uniform in $[0, 63]$ 0.01% constant $2^{63} - 1$	no	63
C3	uniform in $[2^{62}, 2^{62} + 63]$	no	63
C4	uniform in $[2^{47}, 2^{47} + 100\text{K}]$	yes	48
C5	uniform in $[0, 2^{63} - 1]$	no	63

Unless stated otherwise, each column has 128 Mi elements.

fold: (i) the physical byte or bit address corresponding to a logical position must be determined, (ii) one *or more* random accesses are required to obtain all bits of a code word, and (iii) the original data element must be restored from the obtained bits. In the literature, random read access has been investigated to *certain compressed formats* [27, 51, 75]. However, we decided to follow a simple approach by restricting random access to Static BP. For this format, all of the challenges mentioned above can be solved straightforwardly. The integration is accomplished using one specialization of a template function for random read access. Random write access usually addresses the query's result column(s), e.g., in a group-based aggregation. Since these shall be uncompressed anyway, we have not considered random write access to compressed data yet.

Moreover, *on-the-fly de/re-compression* represents the blue print for the realization of our *on-the-fly morphing* operators. In this case, we are able to re-use the *column* and *buffer layer*. But instead of employing (de)compression algorithms, we use *direct morphing algorithms* capable of changing the data representation from one compressed format to another one [19]. In addition, we invoke a specialized operator working directly on compressed data at the *vector register layer*.

5. EXPERIMENTAL EVALUATION

We conducted our experimental evaluation[5] on a machine equipped with an Intel Xeon Gold 6130 clocked at 2.1 GHz. The capacities of the L1, L2, and L3-caches are 32 KiB, 1 MiB, and 22 MiB, respectively. The system has 4 sockets with 32 cores each and exhibits a non-uniform memory access (NUMA). However, we only investigate the single-thread performance, and used `numactl` to ensure that all memory allocations and processing happened on the same socket to exclude NUMA effects [40, 47]. The size of the ECC DDR4 main memory is 384 GiB and all experiments happened entirely in-memory. All our operator and (de)compression algorithm implementations are specialized to an AVX-512 and a scalar version through our TVL [74]. Unless stated otherwise, we report the AVX-512 measurements. We choose a size of 16 KiB, or 2,048 uncompressed data elements, for the internal buffer used at the output side of our on-the-fly de/re-compression operators. Note that this is half of the size of the L1 cache of our machine. We compiled our code using `g++-8.3.0` with the optimization flag `-O3`. The operating system is a 64-bit Ubuntu 18.10 with Linux kernel 4.18.0-13-generic. We repeated all time measurements 10 times and report only the means.

[5] `https://github.com/MorphStore/VLDB-2020`.

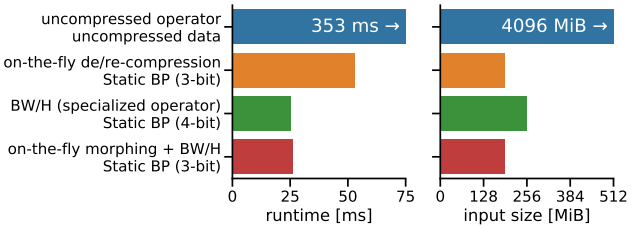

Figure 4: Runtime and input footprint for two state-of-the-art and two novel wrapped `select`-operators.

5.1 Micro-Benchmarks

We start with some mirco-benchmarks to provide a clear understanding of *MorphStore* and, in particular, of our wrapper aproach. The behavior of lightweight integer compression algorithms depends strongly on the data characteristics [18, 20]. Since DBMSs have to handle data with arbitrary properties efficiently, we use synthetic columns C0–C5 and Table 1 summarizes their characteristics. We first investigate the runtime of a single operator as the building block for complex QEPs. In particular, we choose the `select`-operator, which takes a column as input and outputs a sorted column containing the integer *positions* of matching data elements in the input. For each input column, we want to find the (a-priori known) lowest data element in the column using a point predicate.[6] We adapt the distributions mentioned in Table 1 to ensure a certain selectivity.

Illustrative Example. We begin with an example illustrating the usefulness of our novel morphing wrapper. We compare the two state-of-the-art classes of the `select`-operator to our two novel wrapped operators. As the input column, we choose C0 with 512 Mi unsorted data elements in the range $[0, 7]$. We ensure a selectivity of 0.01% to stress the impact of the access to the input data, and leave the output uncompressed for now. Figure 4 shows the runtimes and the footprint of the input column for all four variants.

With purely uncompressed processing, the input column's size is 4 GiB and the selection takes 353 ms. However, C0 can be represented using 3 bits per integer by Static BP, yielding a compressed size of 192 MiB, a reduction by 21.3x. Wrapping the uncompressed operator into on-the-fly de/re-compression reduces the runtime by 6.7x, since much less data has to be loaded from RAM, thereby improving the effective bandwidth. This effect suffices to amortize the computational overhead of on-the-fly decompression.

In the literature, BitWeaving/H (BW/H) was proposed as a specialized operator for a selection scan [51] tailored to the HBP data layout, which is basically Static BP. However, this operator cannot be applied to data packed to 3 bits, because it requires an additional *delimiter bit* in each code word. Thus, we must use Static BP with 4 bits per integer for C0, i.e., we cannot fully exploit the compression potential with the specialized operator. Consequently, the input column grows to 256 MiB, 33% more than before. Our own hand-tuned implementation of the BW/H selection achieves a speedup of 2.1x compared to on-the-fly de/re-compression, although more data has to be transferred

[6] We could as well use a range predicate here, since we assume an order-preserving dictionary encoding for non-integer columns, as stated above.

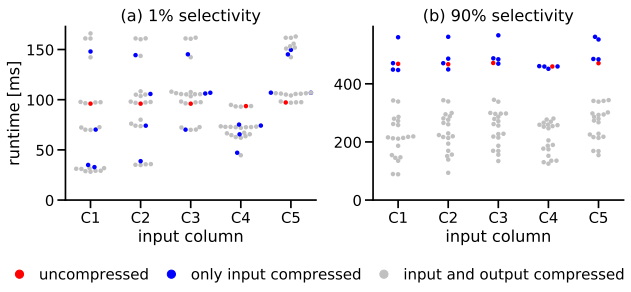

Figure 5: `select` with on-the-fly de/re-compression.

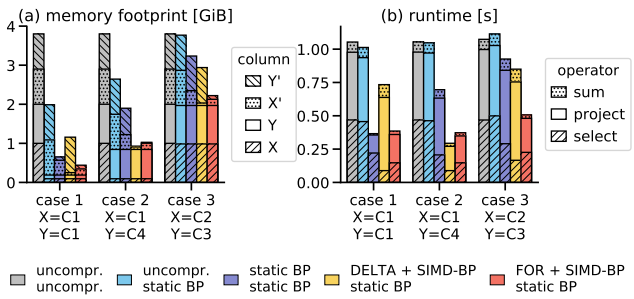

Figure 6: A simple analytical query.

over the memory bus. This is due to the fact that BW/H performs the comparisons directly on compressed data at a much higher data-level and bit-level parallelism.

Finally, our novel on-the-fly morphing provides the freedom to represent the input column with 3 bits per integers *and* to apply BW/H for 4-bit elements by morphing the input from 3 to 4 bits per integer on-the-fly. For this purpose, we employ a highly efficient direct morphing algorithm pipelining morphed data to the BW/H operator at the vector-register granularity.[7] While allowing the lower footprint of 192 MiB for the input column, this on-the-fly morphing operator is only negligible 4% slower than the specialized operator. This is possible, since the reduced transfers from RAM amortize the extra computations for the direct morphing. That is, on-the-fly morphing unifies the superior footprint of on-the-fly de/re-compression and the superior runtime of the specialized operator, which would not be possible without our novel morphing wrapper approach.

While this *example* shows the feasibility and possible benefit of our morphing wrapper, the development of a rich set of direct morphing algorithms is a research topic of its own, and we leave it to future work. In the following, we focus on on-the-fly de/re-compression, since it suffices to realize our design principles and can easily be implemented.

Single on-the-fly De/re-compression Operator. We again focus on the `select`-operator. *MorphStore* currently supports four compressed formats and the uncompressed format. Thus, there are 25 combinations of input and output formats for this operator. Figure 5 gives an overview of the runtimes of *all* combinations for the input columns C1–C5 for selectivities of 1% (a) and 90% (b). For both selectivities, the runtime of the purely uncompressed processing (red dot) is about the same for all input columns, which is expected. With a selectivity of 1%, on-the-fly de/re-compression (blue and gray dots) can decrease the runtime by between 27% (C3) and 71% (C1) in the best case. At the same time, the runtime can increase by up to 73% (C1) in the worst case. As an exception, on-the-fly de/re-compression is *not beneficial* on the hard-to-compress C5. With 90% selectivity, the runtimes can be decreased by between 72% (C3) and 81% (C1) in the best case, but can also be increased by up to 20% in the worst case. Note that compressing also the output column (gray dots) can reduce the runtime much more than compressing only the input column (blue dots). Owing to that, on-the-fly de/re-compression is now also beneficial on C5. This is interesting, since the output column can only be an intermediate in the context of a QEP.

The input format employed by the fastest combination is the same for both selectivities and depends on the input column. For C1, Static BP is preferred, as it contains very small values. C2 contains 0.01% huge outliers, making SIMD-BP512 a better choice, as it can adapt to the local distribution in each block of 512 data values. The distribution of C3 has a narrow range of huge values. Thus, FOR + SIMD-BP512 is most suitable here. DELTA + SIMD-BP512 yields the best runtime for the sorted C4. C5 should always be left uncompressed. The output format employed by the fastest variant is DELTA + SIMD-BP512 in all cases, since the output is always sorted. We conclude that our on-the-fly de/re-compression can reduce the runtime of a single operator *significantly*, if the formats are chosen carefully.

Simple Query. We extend our evaluation to a simple analytical query: given a relation `R` with attributes `X` and `Y`, our query is `SELECT SUM(Y) FROM R WHERE X = c`. The first step of the query execution is the selection we have investigated above, whereby the selectivity is 90%, with input column `X` and output column `X'`. After that, a `project`-operator extracts the data elements at the positions in `X'` from base column `Y` producing intermediate `Y'`. Finally, a `sum`-operator aggregates all data elements in `Y'`. Thus, our simple query consists of three operators accessing two base columns (`X` and `Y`), two intermediates (`X'` and `Y'`), and one result column with a single value, which we ignore below.

We measured the memory footprint and runtime of this query for three cases, each of which is characterized by a certain combination of the base columns in Table 1. Each column can be assigned an individual format and there are $5 \cdot 2 \cdot 5 \cdot 5 = 250$ possible combinations.[8] However, here we concentrate on just a few interesting combinations, while not searching for the best one. Figure 6(a) shows the results for the query's memory footprint, broken down to the individual columns. First, the footprint of the purely uncompressed processing is the same irrespective of the characteristics of the base columns. Applying Static BP for the base columns results in a size reduction to 52% in case 1, since `X` and `Y` contain only very small values here. The other extreme is case 3, where almost no size reduction can be achieved, since both base columns contain data elements of up to 63 bits. If Static BP is applied to the intermediates as well, further reductions to between 17% (case 1) and 85% (case 3) are the consequence. Representing both intermediates using

[7]The detailed description of this algorithm is beyond the scope of this paper. It is included in our open source code.

[8]Random access (performed on column `Y`) is currently only supported for uncompressed data and Static BP.

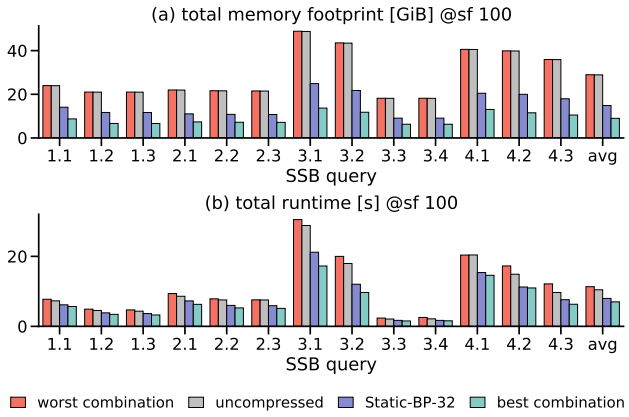

Figure 7: Impact of the format combinations in SSB.

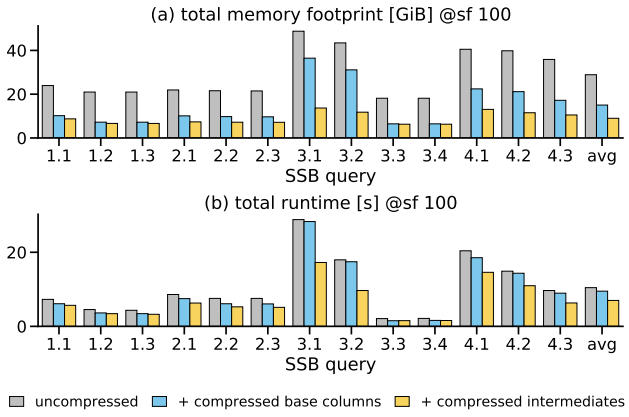

Figure 8: Compressing base data vs. intermediates.

DELTA + SIMD-BP512 reduces the size of `X'` significantly in all cases, since this column is always sorted. At the same time, DELTA + SIMD-BP512 is beneficial for `Y'` only in case 2, where a reduction to 24% is achieved, while in case 1, it even yields a worse memory footprint than Static BP resulting in a reduction to only 30% compared to the uncompressed processing. In cases 1 and 3, `Y'` should rather be represented using FOR + SIMD-BP512 to achieve reductions to 11% and 58%, respectively.

The runtimes of the whole query and the individual operators are displayed in Figure 6(b). Purely uncompressed processing is equally fast in all cases. Applying Static BP only for the base columns can decrease the runtime by only up to 4% (case 1), since writing uncompressed intermediates is very expensive. In case 3, the query runtime is even *increased* by 4%. If the intermediates are compressed as well, the runtimes shrink to between 34% (case 1) and 86% (case 3). While using a suitable cascade for the intermediates could reduce the memory footprint in all cases, the runtimes can only be reduced in cases 2 and 3 by using DELTA or FOR cascaded with SIMD-BP512 in both cases.

We conclude that the continuous compression of both base columns and intermediates can reduce the memory footprint as well as the runtime of a query, if the formats are chosen carefully. Furthermore, given two format combinations, the one that is better with respect to the memory footprint is not always better concerning runtime.

5.2 Star Schema Benchmark (SSB)

Now we investigate the fitness of our novel processing model for complex analytical queries using the SSB [66] at scale factor 100. We applied an order-preserving dictionary coding to all string columns in the schema to obtain integer columns. All 13 queries can be executed on dictionary keys without looking up the strings. This holds for point and range predicates alike. Since MorphStore does not support result sorting yet, we omit the `ORDER BY` clause in all queries.

Impact of Continuous Compression. The QEPs of the SSB queries involve between 6 and 16 base columns and between 13 and 56 intermediates. With our novel processing model, each of the columns can be represented in its individual compressed format, which results in a very high number of possible format combinations. Thus, we first want to find out the impact of different format combinations and the improvement through the optimal format combination. We allow compression for base columns and intermediates here. We consider the following format combinations for each query: (i) purely uncompressed, (ii) Static BP with a fixed bit width of 32 bits per data element for all columns (Static-BP-32), a simple compression approach not taking individual data characteristics into account, (iii) the actual best format combination, where each column can have its individual format, which only becomes possible with our novel morphing wrapper, and (iv) the actual worst format combination. We determined the best/worst combination regarding memory footprint by exhaustively trying each available format for each column *individually* (since column footprints add up) and employing the format yielding the lowest/highest physical size. Concerning runtime, we applied a greedy strategy which, starting at the base data, considers one column at a time by trying all available formats for that column, measuring the resulting query runtimes and fixing the column's format to the one yielding the best/worst runtime for the next steps of the search. Note that these best/worst combinations are allowed to employ the uncompressed format. The best format combinations usually mix all available formats within a single query. However, due to a lack of space, we do not present them here.

Figure 7(a) shows the results for the memory footprint. The purely uncompressed processing achieves by far the worst memory footprints for all queries. Using Static-BP-32 for all columns reduces the memory footprint to about 50% for all queries. However, employing the optimal format for each column yields a reduction of the footprint to between 27% (Q3.2) and 36% (Q1.1) (31% on average). Figure 7(b) depicts the query runtimes. Here, the worst combination results in a runtime *increase* by 8% on average, compared to the purely uncompressed case. Employing Static-BP-32 reduces the runtimes to between 67% (Q3.2) and 85% (Q1.2) (76% on average). However, the best format combinations yield runtime reductions to 54% and 76% for these queries (67% on average), compared to purely uncompressed processing. To sum up, the continuous use of compression can significantly reduce memory footprint and runtime, whereby optimal results require a column-specific format selection enabled only through our novel morphing wrapper.

Compressing Base Data vs. Intermediates. Next, we evaluate in how far our continuous compression of *inter-*

mediates contributes to these improvements, compared to the already established compression of *base data*. We start by not allowing compression at all, then we allow compression for base columns only, and, finally, also for intermediates, whereby we use the actual best format combinations. The resulting memory footprints can be found in Figure 8(a). Compressed base data already reduces the footprints significantly to between 34% (Q1.3) and 75% (Q3.1) (52% on average). The impact of offering compression also for intermediates is highly query-dependent. For Q3.3 and 3.4, virtually no further improvement is possible, while a further reduction to 27% can be achieved for Q3.2. On average, compressing intermediates reduces the footprints to 31%, i.e., 21% further than with base columns only. Figure 8(b) depicts the runtime results. These can be reduced to up to 73% (Q3.3) when compressing only base data (91% on average), while the additional compression of intermediates achieves a reduction to up to 54% (Q3.2) (67% on average). We can conclude that our continuous compression of intermediates contributes *significantly* to the overall memory and runtime reductions achievable through compression. Regarding the runtime, the intermediates' impact is even higher than that of base columns, on average.

Comparison to MonetDB. Next, we move on to a comparison to MonetDB, the system that is closest to our processing model *in the uncompressed case*. To ensure a fair comparison, we compiled MonetDB-11.31.13 with all relevant optimization switches on using the same compiler (and version) and run it on the same machine as *MorphStore*. We run MonetDB also in single-threaded and read-only mode. Although MonetDB supports string columns, we use the same dictionary-encoded base data. For our experiments, the QEPs used in *MorphStore* imitate those of MonetDB closely, including the same join order. We used MonetDB's internal tools for measuring the mere *run*time (excluding the time spent on query optimization) of each query 12 times, and discarded the first two measurements for each query.

The runtimes are given in Figure 9. Since MonetDB supports neither vectorization nor compressed intermediates, we first investigate the scalar execution on purely uncompressed data (all columns use 64-bit) for a fair comparison. None of the systems is faster than any other for all queries, but on average, MorphStore is 1.4x faster. MorphStore also supports a state-of-the-art vectorized execution using AVX-512 and this reduces the runtime by up to 54% (Q1.3) or increases it by up to 14% (Q4.1) (on average, it decreases by 11%). However, maximum performance is only provided by our novel continuous compression for base columns and intermediates according to the ideal format combination, as this reduces the runtimes further to between 35% (Q1.3) and 82% (Q4.1) of the scalar uncompressed processing in *MorphStore* (60% on average). While MonetDB has no explicit support for compression of intermediates, we try to simulate it by using the narrowest integer type possible for all base data columns in MonetDB. We can see that this improves the runtime of MonetDB to 90% of its uncompressed runtime, on average, however, this is still much slower than *MorphStore's holistic compression-enabled* approach. We conclude that, even without our proposed continuous compression of intermediates, MorphStore is slightly faster than a state-of-the-art system adopting the operator-at-a-time processing model, which proves the general quality of our implementation. Moreover, our *holistic compression-enabled* approach accelerates SSB queries by 40%, on average, compared to a scalar processing of uncompressed data.

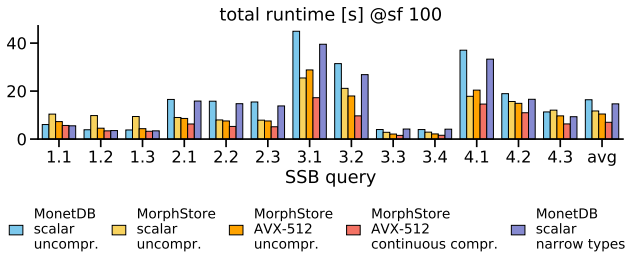

Figure 9: Comparing MonetDB and *MorphStore*.

6. ENHANCING QUERY OPTIMIZATION

Our evaluation revealed that the continuous compression of intermediates can improve or worsen the memory footprint and query runtime, depending on the combination of the intermediates' formats. Thus, the reasonable selection of these formats is a new subject for query optimization. Compression-aware query optimization has already been discussed for queries over bit vectors [31] and for row-stores [16, 76], but these approaches are not applicable to our continuous compression, since we significantly advance the use of compression during query execution. Nevertheless, inspired by *Two-Step* [16], in the following, we show how an existing query optimizer can be extended by a compression-aware *secondary* optimization phase. While a secondary step might fail to find the globally optimal QEP, it can yield an improvement compared to the state-of-the-art.

We assume that a classical query optimizer provides (i) a QEP with uncompressed intermediates and (ii) some basic data characteristics of all intermediates. We focus on the selection of suitable formats for *on-the-fly de/re-compression*. In our previous work, we proposed and evaluated a cost model for selecting a suitable lightweight integer compression algorithm for a given column [20]. Based on basic column characteristics, such as the number of data elements, a histogram, and the number of distinct values, our cost model can estimate the compression rate as well as (de)compression runtime of a lightweight integer compression algorithm.

A compression-aware query optimization could aim for two objectives: (i) minimizing the memory footprint or (ii) minimizing the query runtime. *Minimizing the memory footprint:* The size of each column can be minimized individually by estimating its compressed size in all available formats using our cost model and choosing the format yielding the lowest size. *Minimizing the query runtime:* Our morphing-wrapper does not change the internal processing of the wrapped operators. Thus, we only need to model the cost of the wrapper to compare two alternative format combinations. Each column in a QEP has to be compressed by the operator producing it (unless it is a base column) and decompressed by each operator consuming it. Using our cost model, we can estimate the runtimes of these (de)compressions, whose sum constitutes the share of the considered column in the overall wrapper cost in the QEP. This sum can be minimized for each column individually.

We applied these optimization strategies to the SSB. Figure 10(a) reveals that we always achieve the perfect mem-

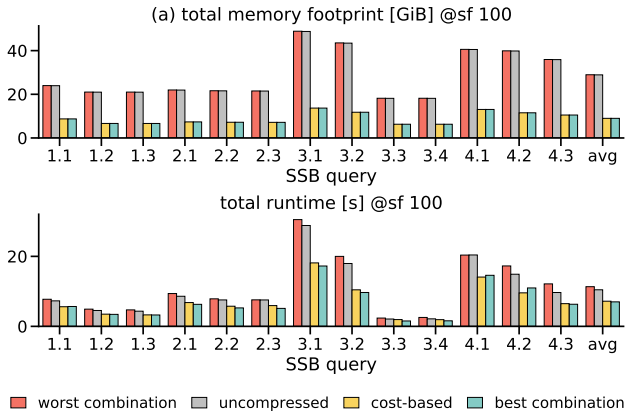

Figure 10: Fitness of our cost-based optimization strategies for on-the-fly de/re-compression.

ory footprint. Figure 10(b) shows that the query runtime achieved with our cost-based optimization strategy is only up to 26% (Q3.3) worse than the runtime yielded by the greedily determined format combination (3% on average). Interestingly, our cost-based selection is even 13% faster than the combination determined by the greedy algorithm for Q4.2. For both objectives and all queries, our cost-based optimization strategy (i) yields an actual improvement compared to the uncompressed baseline, and (ii) successfully avoids the worst format combination.

7. DISCUSSION

In this paper, we focused on a single-threaded, in-memory columnar query processing. On the one hand, we are confident that our novel model can be applied to column-stores in general, since only the set of relevant compression algorithms differs depending on the storage medium [2, 13, 44, 80]. While memory-centric systems use *lightweight* integer compression schemes to optimize the main-memory access bottleneck, so-called *heavyweight* generic compression schemes such as Huffmann [35], Lempel-Ziv [78, 84], or Snappy [29] are used to optimize the disk access bottleneck in disk-centric environments. The main difference between heavy- and lightweight compression algorithms is their computational complexity. However, our approach is independent of any specific compression scheme. Thus, it should be able to embrace such heavyweight algorithms, too.

On the other hand, our novel model could be used as a foundation in multi-threaded settings. As shown in several papers, the *data-oriented database architecture* provides a superior scalability on common scale-up hardware [42, 41, 54, 62]. In contrast to the traditional transaction-oriented database architecture, transactions are not first-class citizens anymore and are processed in a distributed fashion by worker threads. All data objects, e.g. columns, are partitioned and partitions are exclusively processed by the assigned worker thread that is pinned to a specific hardware thread. That means, each single worker thread can process one specific column partition with our novel processing model. The communication between worker threads during query processing is handled via a message passing layer to exchange intermediate results. From that perspective, the combination of our novel processing model with the data-oriented architecture should be explored in the near future. This combination offers several advantages like (i) each column partition could be compressed with a specific compression schema or (ii) the intermediates could be compressed with specific schemes to optimize the communication.

Boncz et al. [14, 85] identified that the full materialization of intermediates performed by the operator-at-a-time model makes query processing memory-bound if the intermediates' sizes exceed that of the cache. To address this issue, the authors present the *X100* query engine with the *vector-at-a-time* model combining the column-wise processing of the operator-at-a-time model with a pipelined operator execution. There, operators consume and produce blocks of a column, so-called *vectors*. Our *holistic compression-enabled processing model* could be extended to this concept by invoking the input-side morphing wrapper for each compressed input *vector* and pipelining compressed output *vectors* directly to the next operator. By employing specialized operators inside the morphing-wrapper, *vectors* could be processed with a higher bit or data-level parallelism to improve efficiency beyond a mere increase of the effective bandwidth.

An orthogonal approach to optimize query processing is to avoid intermediate results [43, 53, 55]. For example, Neumann introduced an approach by maintaining the pipeline processing from one operator to the next [53]. Since these pipelines are query-dependent, they must be compiled at query run-time. Nevertheless, the materialization of intermediate results is still required at the pipeline boundaries, for instance, when building the hash table of a hash join. The DBMS *Peloton* [55] employs an advancement of this approach that was proposed by Menon et al. [52]. The authors argue that the strict avoidance of intermediates and the pipeline processing disallow the exploitation of inter-tuple-parallelism offered by modern microprocessors. In particular, the authors show how to leverage SIMD and prefetching instructions. Moreover, they perform a strategic partial materialization of selected intermediate results, while they otherwise avoid intermediates using pipelining. An interesting research question would be to investigate our novel model for the optimization of these partial materializations.

8. CONCLUSION

We presented *MorphStore*, an in-memory analytical query engine with a novel *holistic compression-enabled* processing model. We optimize the *operator-at-a-time* processing model by establishing the continuous use of lightweight integer compression for all intermediate results in a query execution plan. Our approach is based on a novel *morphing wrapper* embracing and enhancing existing query operators for (un)compressed data and allowing a change of the compressed format during query processing to adapt to changing data characteristics. That way, we are able to *significantly* reduce the memory footprint and runtime of complex analytical queries. Moreover, we complement our novel processing model with novel compression-aware optimization strategies selecting suitable compressed formats for all intermediates.

Acknowledgment

This work was partly funded by the German Research Foundation (DFG) via LE-1416/26-1 and via RTG 1907 (RoSI).

9. REFERENCES

[1] D. Abadi, P. A. Boncz, S. Harizopoulos, S. Idreos, and S. Madden. The design and implementation of modern column-oriented database systems. *Foundations and Trends in Databases*, 5(3):197–280, 2013.

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

[3] V. N. Anh and A. Moffat. Inverted index compression using word-aligned binary codes. *Inf. Retr.*, 8(1):151–166, 2005.

[4] V. N. Anh and A. Moffat. Index compression using 64-bit words. *Softw., Pract. Exper.*, 40(2), 2010.

[5] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *PVLDB*, 7(1):85–96, 2013.

[6] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. Main-memory hash joins on modern processor architectures. *IEEE Trans. Knowl. Data Eng.*, 27(7):1754–1766, 2015.

[7] 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.

[8] C. Binnig, S. Hildenbrand, and F. Färber. Dictionary-based order-preserving string compression for main memory column stores. In *SIGMOD*, pages 283–296, 2009.

[9] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. In *SIGMOD*, pages 37–48, 2011.

[10] M. Boissier. Reducing the footprint of main memory HTAP systems: Removing, compressing, tiering, and ignoring data. In *PhD@VLDB*, volume 2175 of *CEUR Workshop Proceedings*, 2018.

[11] M. Boissier and M. Jendruk. Workload-driven and robust selection of compression schemes for column stores. In *EDBT*, pages 674–677, 2019.

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

[13] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the memory wall in monetdb. *Commun. ACM*, 51(12):77–85, 2008.

[14] P. A. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, pages 225–237, 2005.

[15] S. Chaudhuri, U. Dayal, and V. R. Narasayya. An overview of business intelligence technology. *Commun. ACM*, 54(8):88–98, 2011.

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

[17] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y. Chen, A. Baransi, S. Kumar, and P. Dubey. Efficient implementation of sorting on multi-core SIMD CPU architecture. *PVLDB*, 1(2):1313–1324, 2008.

[18] 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.

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

[20] P. Damme, A. Ungethüm, J. Hildebrandt, D. Habich, and W. Lehner. From a comprehensive experimental survey to a cost-based selection strategy for lightweight integer compression algorithms. *ACM Trans. Database Syst.*, 44(3):9:1–9:46, 2019.

[21] D. Das, J. Yan, M. Zaït, S. R. Valluri, N. Vyas, R. Krishnamachari, P. Gaharwar, J. Kamp, and N. Mukherjee. Query optimization in oracle 12c database in-memory. *PVLDB*, 8(12):1770–1781, 2015.

[22] R. Delbru, S. Campinas, and G. Tummarello. Searching web data: An entity retrieval and high-performance indexing model. *J. Web Semant.*, 10:33–58, 2012.

[23] M. Dreseler, J. Kossmann, M. Boissier, S. Klauck, M. Uflacker, and H. Plattner. Hyrise re-engineered: An extensible database system for research in relational in-memory data management. In *EDBT*, pages 313–324, 2019.

[24] P. Elias. Universal codeword sets and representations of the integers. *IEEE Trans. Information Theory*, 21(2):194–203, 1975.

[25] F. Faerber, A. Kemper, P. Larson, J. J. Levandoski, T. Neumann, and A. Pavlo. Main memory database systems. *Foundations and Trends in Databases*, 8(1-2):1–130, 2017.

[26] Z. Feng and E. Lo. Accelerating aggregation using intra-cycle parallelism. In *ICDE*, pages 291–302, 2015.

[27] 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.

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

[29] Google. Snappy: A fast compressor/decompressor. `https://github.com/google/snappy`.

[30] G. Graefe and L. D. Shapiro. Data compression and database performance. In *Symposium on Applied Computing*, pages 22–27, 1991.

[31] G. Guzun and G. Canahuate. Hybrid query optimization for hard-to-compress bit-vectors. *VLDB J.*, 25(3):339–354, 2016.

[32] D. Habich, P. Damme, A. Ungethüm, J. Pietrzyk, A. Krause, J. Hildebrandt, and W. Lehner. Morphstore - in-memory query processing based on morphing compressed intermediates LIVE. In *SIGMOD*, pages 1917–1920, 2019.

[33] J. Hildebrandt, D. Habich, P. Damme, and W. Lehner. Compression-aware in-memory query processing: Vision, system design and beyond. In *ADMS@VLDB*, pages 40–56, 2016.

[34] J. Hildebrandt, D. Habich, T. Kühn, P. Damme, and W. Lehner. Metamodeling lightweight data compression algorithms and its application scenarios. In *ER Forum*, pages 128–141, 2017.

[35] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the*

Institute of Radio Engineers, 40(9), 1952.

[36] C. J. Hughes. *Single-Instruction Multiple-Data Execution*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2015.

[37] 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. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.

[38] 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.

[39] T. Kersten, V. Leis, A. Kemper, T. Neumann, A. Pavlo, and P. A. Boncz. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *PVLDB*, 11(13):2209–2222, 2018.

[40] T. Kiefer, B. Schlegel, and W. Lehner. Experimental evaluation of NUMA effects on database management systems. In *BTW*, pages 185–204, 2013.

[41] T. Kissinger, D. Habich, and W. Lehner. Adaptive energy-control for in-memory database systems. In *SIGMOD*, pages 351–364, 2018.

[42] T. Kissinger, T. Kiefer, B. Schlegel, D. Habich, D. Molka, and W. Lehner. ERIS: A numa-aware in-memory storage engine for analytical workload. In *ADMS@VLDB*, pages 74–85, 2014.

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

[44] M. Kornacker et al. Impala: A modern, open-source SQL engine for hadoop. In *CIDR*, 2015.

[45] 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.

[46] 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.

[47] V. Leis, P. A. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age. In *SIGMOD*, pages 743–754, 2014.

[48] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Softw., Pract. Exper.*, 45(1):1–29, 2015.

[49] D. Lemire and O. Kaser. Reordering columns for smaller indexes. *Inf. Sci.*, 181(12):2550–2570, 2011.

[50] D. Lemire, N. Kurz, and C. Rupp. Stream vbyte: Faster byte-oriented integer compression. *Inf. Process. Lett.*, 130:1–6, 2018.

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

[52] P. Menon, A. Pavlo, and T. C. Mowry. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *PVLDB*, 11(1):1–13, 2017.

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

[54] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *PVLDB*, 3(1):928–939, 2010.

[55] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, T. C. Mowry, M. Perron, I. Quah, S. Santurkar, A. Tomasic, S. Toor, D. V. Aken, Z. Wang, Y. Wu, R. Xian, and T. Zhang. Self-driving database management systems. In *CIDR*, 2017.

[56] J. Pietrzyk, A. Ungethüm, D. Habich, and W. Lehner. Fighting the duplicates in hashing: Conflict detection-aware vectorization of linear probing. In *BTW*, pages 35–53, 2019.

[57] J. Plaisance, N. Kurz, and D. Lemire. Vectorized vbyte decoding. *CoRR*, abs/1503.07387, 2015.

[58] O. Polychroniou, A. Raghavan, and K. A. Ross. Rethinking SIMD vectorization for in-memory databases. In *SIGMOD*, pages 1493–1508, 2015.

[59] O. Polychroniou and K. A. Ross. A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort. In *SIGMOD*, pages 755–766, 2014.

[60] O. Polychroniou and K. A. Ross. Vectorized bloom filters for advanced SIMD processors. In *DaMoN@SIGMOD*, pages 6:1–6:6, 2014.

[61] O. Polychroniou and K. A. Ross. Towards practical vectorized analytical query engines. In *DaMoN@SIGMOD*, pages 10:1–10:7, 2019.

[62] D. Porobic, E. Liarou, P. Tözün, and A. Ailamaki. Atrapos: Adaptive transaction processing on hardware islands. In *ICDE*, pages 688–699, 2014.

[63] V. Raman, G. K. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, T. Malkemus, R. Müller, I. Pandis, B. Schiefer, D. Sharpe, R. Sidle, A. J. Storm, and L. Zhang. DB2 with BLU acceleration: So much more than just a column store. *PVLDB*, 6(11):1080–1091, 2013.

[64] R. Rice and J. Plaunt. Adaptive variable-length coding for efficient compression of spacecraft television data. *IEEE Transactions on Communication Technology*, 19(6):889–897, 1971.

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

[66] J. Sanchez. A review of star schema benchmark. *CoRR*, abs/1606.00295, 2016.

[67] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey. Fast sort on cpus and gpus: a case for bandwidth oblivious SIMD sort. In *SIGMOD*, pages 351–362, 2010.

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

[69] F. Silvestri and R. Venturini. Vsencoding: efficient coding and fast decoding of integer lists via dynamic programming. In *CIKM*, pages 1219–1228, 2010.

[70] sjoerd. MonetDB goes headless MonetDB blog. `https://www.monetdb.org/blog/monetdb-goes-headless`, December 2016. Accessed: 2020-02-29.

[71] A. A. Stepanov, A. R. Gangolli, D. E. Rose, R. J. Ernst, and P. S. Oberoi. Simd-based decoding of posting lists. In *CIKM*, pages 317–326, 2011.

[72] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-store: A column-oriented DBMS. In *VLDB*, pages 553–564, 2005.

[73] A. Ungethüm, J. Pietrzyk, P. Damme, D. Habich, and W. Lehner. Conflict detection-based run-length encoding - AVX-512 CD instruction set in action. In *ICDE Workshops*, pages 96–101, 2018.

[74] A. Ungethüm, J. Pietrzyk, P. Damme, A. Krause, D. Habich, W. Lehner, and E. Focht. Hardware-oblivious SIMD parallelism for in-memory column-stores. In *CIDR*, 2020.

[75] J. Wang, C. Lin, R. He, M. Chae, Y. Papakonstantinou, and S. Swanson. MILC: inverted list compression in memory. *PVLDB*, 10(8):853–864, 2017.

[76] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte. The implementation and performance of compressed databases. *SIGMOD Record*, 29(3):55–67, 2000.

[77] 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.

[78] R. N. Williams. An extremely fast ziv-lempel data compression algorithm. In *DCC*, pages 362–371, 1991.

[79] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *WWW*, pages 401–410, 2009.

[80] M. Zarubin, P. Damme, T. Kissinger, D. Habich, W. Lehner, and T. Willhalm. Integer compression in nvram-centric data stores: Comparative experimental analysis to DRAM. In *DaMoN@SIGMOD*, pages 11:1–11:11, 2019.

[81] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *WWW*, pages 387–396, 2008.

[82] 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 Trans. Inf. Syst.*, 33(3):15:1–15:28, 2015.

[83] J. Zhou and K. A. Ross. Implementing database operations using SIMD instructions. In *SIGMOD*, pages 145–156, 2002.

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

[85] M. Zukowski, P. A. Boncz, N. Nes, and S. Héman. Monetdb/x100 - A DBMS in the CPU cache. *IEEE Data Eng. Bull.*, 28(2):17–22, 2005.

[86] M. Zukowski, S. Héman, N. Nes, and P. A. Boncz. Super-scalar RAM-CPU cache compression. In *ICDE*, page 59, 2006.