Dieses Dokument ist eine Zweitveröffentlichung (Postprint) /

This is a self-archiving document (accepted version):

Dirk Habich, Patrick Damme, Annett Ungethüm, Johannes Pietrzyk, Alexander Krause, Juliana Hildebrandt, Wolfgang Lehner

MorphStore - In-Memory Query Processing based on Morphing Compressed Intermediates LIVE

Erstveröffentlichung in / First published in:

SIGMOD/PODS '19: International Conference on Management of Data, Amsterdam 30.06. – 05.07.2019. ACM Digital Library, S. 1917-1920. ISBN 978-1-4503-5643-5

DOI: https://doi.org/10.1145/3299869.3320234

Diese Version ist verfügbar / This version is available on:

https://nbn-resolving.org/urn:nbn:de:bsz:14-qucosa2-806340

SLUB
Wir führen Wissen.

TECHNISCHE UNIVERSITÄT DRESDEN

QUCOSA
Quality Content of Saxony

Final edited form was published in "SIGMOD/PODS '19: International Conference on Management of Data. Amsterdam 2019", S.1917-1920, ISBN 978-1-4503-5643-5
https://doi.org/10.1145/3299869.3320234

MorphStore — In-Memory Query Processing based on Morphing Compressed Intermediates LIVE

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

ABSTRACT

In this demo, we present *MorphStore*, an in-memory column store with a novel compression-aware query processing concept. Basically, compression using lightweight integer compression algorithms already plays an important role in existing in-memory column stores, but mainly for base data. The continuous handling of compression from the base data to the intermediate results during query processing has already been discussed, but not investigated in detail since the computational effort for compression as well as decompression is often assumed to exceed the benefits of a reduced transfer cost between CPU and main memory. However, this argument increasingly loses its validity as we are going to show in our demo. Generally, our novel compression-aware query processing concept is characterized by the fact that we are able to speed up the query execution by morphing compressed intermediate results from one scheme to another scheme to dynamically adapt to the changing data characteristics during query processing. Our morphing decisions are made using a cost-based approach.

CCS CONCEPTS

• **Information systems → DBMS engine architectures**; **Query optimization**; **Query operators**; **Main memory engines**; *Query planning*.

This work was partly funded (1) by the German Research Foundation (DFG) within the CRC 912 (HAEC), RTG 1907 (RoSI) as well as by an individual project LE-1416/26-1, (2) by the German Federal Ministry of Education and Research (BMBF) with the EXPLOIDS project (16KIS0523), and (3) by NEC.

©2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *SIGMOD '19, June 30-July 5, 2019, Amsterdam, Netherlands*
https://doi.org/10.1145/3299869.3320234

KEYWORDS

in-memory column store; lightweight integer compression; query processing; vectorization

ACM Reference Format:
Dirk Habich, Patrick Damme, Annett Ungethüm, Johannes Pietrzyk, Alexander Krause, Juliana Hildebrandt, Wolfgang Lehner. 2019. MorphStore — In-Memory Query Processing based on Morphing Compressed Intermediates LIVE. In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3299869.3320234

1 INTRODUCTION

In-memory database systems pursue a main memory-centric architecture approach and assume that all relevant data (base data as well as intermediates) can be fully kept in the main memory of a computer or of a computer network [8]. For OLAP workloads, in-memory column store systems are perfectly suited, because relational tables are organized by column rather than by row and based on that, queries only need to read relevant data columns [1, 8]. In these systems, lightweight integer compression algorithms play an important role [1, 13]. Aside from reducing the amount of data, compressed data offers several advantages such as less time spent on load and store instructions and a better utilization of the cache hierarchy. Moreover, a direct processing of the compressed data is possible in many cases. As we have shown in [6, 7], there is a large variety of lightweight integer compression schemes available and there is no single-best algorithm, but the decision depends on data as well as on hardware properties. However, existing systems only provide a very limited set of compression algorithms for base data [1, 8, 13]. 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. Thus, the full optimization potential is not exploited.

To overcome that, we developed *MorphStore*[1], a regular in-memory column store with a novel compression-aware query

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

Provided by Sächsische Landesb bliothek, Staats- und Universitätsbibliothek Dresden

Final edited form was published in "SIGMOD/PODS '19: International Conference on Management of Data. Amsterdam 2019", S.1917-1920, ISBN 978-1-4503-5643-5
https://doi.org/10.1145/3299869.3320234

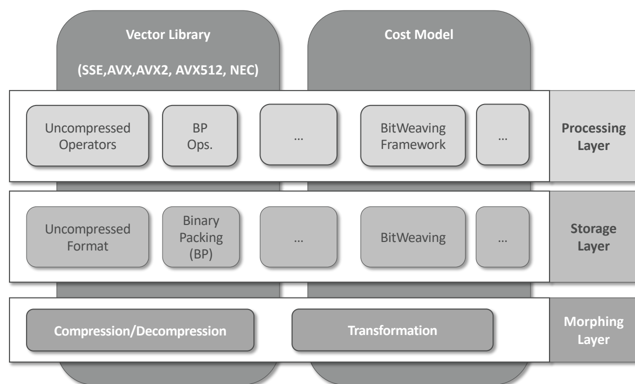

Figure 1: *MorphStore* architecture.

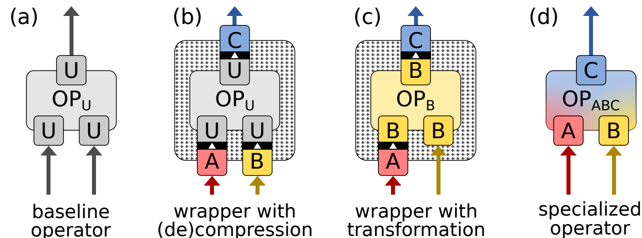

Figure 2: Integration of compression and operators. A **to** C **are compressed formats;** U **is uncompressed [4].**

processing concept [4]. The unique features of *MorphStore* are: (i) support of a large variety of lightweight integer compression algorithms, (ii) a continuous handling of compression from base data through intermediate results, (iii) a cost-based decision for the best-suited compression algorithm, and (iv) morphing intermediates from one to another compression scheme to dynamically adapt the physical representation to the changing data characteristics at query run-time.

2 ARCHITECTURE

Figure 1 depicts a high-level architecture of *MorphStore* which is explained in the following in more detail.

Storage Layer. This layer follows a well-known approach: (i) encode values of each column as a sequence of integers using some kind of dictionary encoding [2] and (ii) apply lightweight lossless integer compression to each sequence of integers resulting in a sequence of compressed column codes [1, 4, 6]. As illustrated in Figure 1, *MorphStore* does not assume or prefer a specific in-memory storage layout. Instead, it aims to support a large variety of lightweight integer compression *algorithms* and a variety of specific *layouts* for compressed data, e.g. BitWeaving [16]. In principle, these are two different things, but since some compression algorithms also specify a storage layout for the compressed data, the pool of possible layouts becomes even larger. Thus, this layer focuses on the different layouts for storing uncompressed as well as compressed sequences of integers in-memory.

We follow this approach, since, as we have shown in [6], the compression algorithms are always tailored to certain data characteristics and their behavior in terms of performance and compression ratio depends strongly on the data. There is no single-best compression algorithm [6], thus we need a large variety to support all possible data characteristics. For the algorithm selection, we introduced a compression-specific cost model allowing the estimation of the compression ratio as well as the performance in [7].

Morphing Layer. While the *storage layer* focuses on providing different layouts, the *morphing layer* provides an infrastructure for a seamless transition (we call it morphing) from data in a specific layout into another layout. Thus, the different (de)compression algorithms, which are responsible for transforming uncompressed data into the corresponding compressed layout and vice versa, are major parts of this layer. Furthermore, we introduced novel transformation algorithms in [5] to directly transform data from a compressed source layout into a compressed target layout. These transformations are also components of this layer.

For our morphing purposes—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 (also called vector 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 the lightweight integer compression domain in recent years [6, 9, 15, 21]. To support the different available vector extensions as well as a dedicated vector processor provided by NEC [12, 19] with a low effort, we developed a *Vector Library* abstracting different SIMD extensions, which is comparable to the approach of [20]. This library is a core component of *MorphStore*.

Processing Layer. The execution model of *MorphStore* corresponds to column-at-a-time, where all intermediates are materialized in main memory. Thus, this layer provides all physical query operators for *MorphStore*, thereby different degrees of integration between these operators and compression are possible. Figure 2 shows these variants and *MorphStore* supports all of them. The selection of the best-suited operator variant within a query execution plan (QEP) will be done using an appropriate cost model in a subsequent step to the regular query optimization. Figure 2(a) shows the baseline variant of processing only uncompressed data. In the following, we assume we want to support n compressed formats for *one* operator.

Provided by Sächsische Landesb bliothek, Staats- und Universitätsbibliothek Dresden

Final edited form was published in "SIGMOD/PODS '19: International Conference on Management of Data. Amsterdam 2019", S.1917-1920, ISBN 978-1-4503-5643-5
https://doi.org/10.1145/3299869.3320234

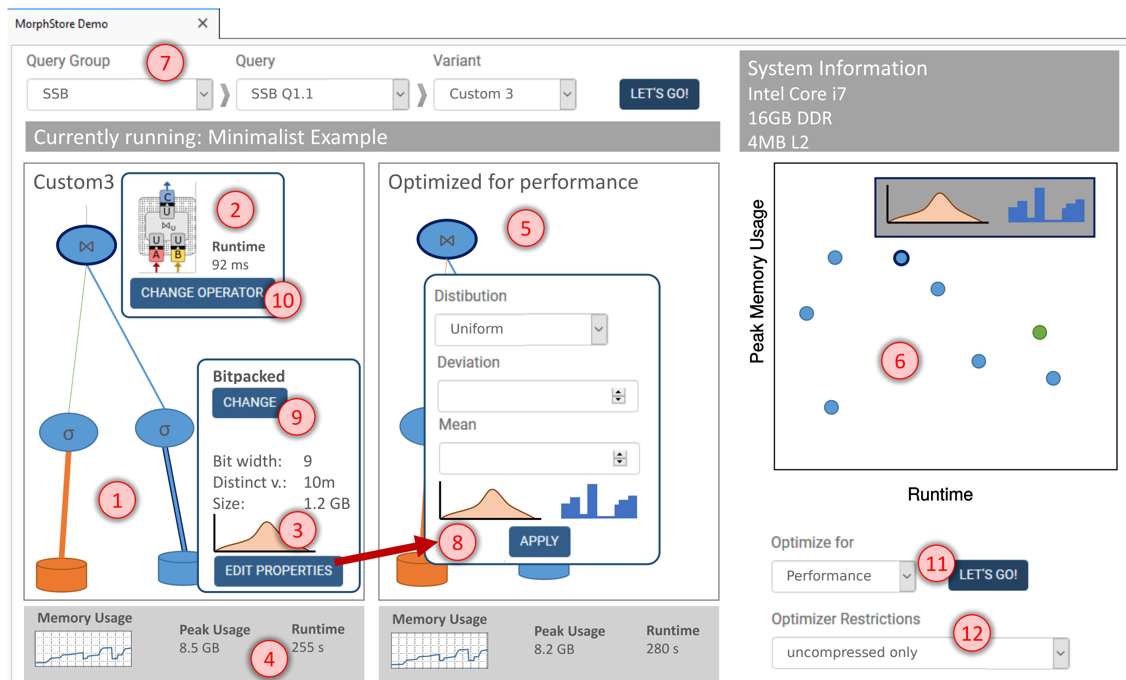

Figure 3: The interactive graphical user interface of our demonstration.

A first variant to support compressed intermediates is shown in Figure 2(b). 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 [3], but to the best of our knowledge, it has never been investigated in practice. For efficiency, in *MorphStore* the decompression (recompression) does 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 variant 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 variant is to adapt the operator such that it can work *directly* on compressed data (Figure 2(c)). Existing works such as [14, 16] have already proposed *certain* operators on *certain* compressed formats. We contribute to this line of research by covering the formats of recent vectorized compression algorithms. For this variant, we assume a common compression format (format B in Figure 2(c)) 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. The idea of bringing compressed inputs into a common format has already been proposed in [14], but only for joins on dictionary encoded data – and without *direct* transformations. 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 variant maximizes the efficiency by tailoring the operator to a specific combination of formats (Figure 2(d)). Unfortunately, this approach implies the highest implementation effort, requiring n^{i+o} operator variants.

3 DEMONSTRATION DETAILS

The overall aim is to present *MorphStore* and its compression-aware query processing concept. For this, we address two aspects in our demo: (1) we would like to convey to the attendee an understanding of how changing the compressed layout of the data during query execution impacts the two important optimization objectives runtime and memory usage, and (2) we introduce our *MorphStore* design to show that we are able to select a *good* compression-aware query execution plan (QEP) with respect to these two objectives.

We provide an interactive graphical user interface for the detailed *visualization* and *comparison* of multiple compression-aware QEP configurations as shown in Figure 3. Each QEP is visualized as a tree whose nodes represent compression-aware physical operators and whose edges represent compressed base data or intermediates (1) (bracket in Figure 3). For a first overview, the edges are colored depending on the

Provided by Sächsische Landesb bliothek, Staats- und Universitätsbibliothek Dresden

Final edited form was published in "SIGMOD/PODS '19: International Conference on Management of Data. Amsterdam 2019", S.1917-1920, ISBN 978-1-4503-5643-5
https://doi.org/10.1145/3299869.3320234

we want to integrate the architecture of *MorphStore* with the *DORA* architecture [18] to efficiently support scale-up hardware systems. Furthermore, we are extending our research activities to reduce the implementation effort required to support the large variety of lightweight integer compression algorithms [10, 11].

REFERENCES

[1] D. Abadi et al. The design and implementation of modern column-oriented database systems. *Foundations and Trends in Databases*, 5(3):197–280, 2013.

[2] C. Binnig et al. Dictionary-based order-preserving string compression for main memory column stores. In *SIGMOD*, pages 283–296, 2009.

[3] Z. Chen et al. Query optimization in compressed database systems. In *SIGMOD*, pages 271–282, 2001.

[4] P. Damme. Query processing based on compressed intermediates. In *VLDB PhD Workshop*, 2017.

[5] P. Damme et al. Direct transformation techniques for compressed data: General approach and application scenarios. In *ADBIS*, pages 151–165, 2015.

[6] P. Damme et al. Lightweight data compression algorithms: An experimental survey (experiments and analyses). In *EDBT*, pages 72–83, 2017.

[7] P. Damme et al. From a comprehensive experimental survey to a cost-based selection strategy for lightweight integer compression algorithms. *Appears in TODS (accepted)*, 2019.

[8] F. Faerber et al. Main memory database systems. *Foundations and Trends in Databases*, 8(1-2):1–130, 2017.

[9] D. Habich et al. Make larger vector register sizes new challenges?: Lessons learned from the area of vectorized lightweight compression algorithms. In *DBTest@SIGMOD*, pages 8:1–8:6, 2018.

[10] J. Hildebrandt et al. Compression-aware in-memory query processing: Vision, system design and beyond. In *IMDM@VLDB*, pages 40–56, 2016.

[11] J. Hildebrandt et al. Metamodeling lightweight data compression algorithms and its application scenarios. In *ER Forum*, pages 128–141, 2017.

[12] K. Komatsu et al. Performance evaluation of a vector supercomputer SX-Aurora TSUBASA. In *SC*, pages 54:1–54:12, 2018.

[13] H. Lang et al. Data blocks: Hybrid OLTP and OLAP on compressed storage using both vectorization and compilation. In *SIGMOD*, pages 311–326, 2016.

[14] J. Lee et al. Joins on encoded and partitioned data. *PVLDB*, 7(13):1355–1366, 2014.

[15] D. Lemire et al. Decoding billions of integers per second through vectorization. *Softw., Pract. Exper.*, 45(1):1–29, 2015.

[16] Y. Li et al. Bitweaving: fast scans for main memory data processing. In *SIGMOD*, pages 289–300, 2013.

[17] P. E. O'Neil et al. The star schema benchmark and augmented fact table indexing. In *TPCTC*, pages 237–252, 2009.

[18] I. Pandis et al. Data-oriented transaction execution. *PVLDB*, 3(1):928–939, 2010.

[19] J. Pietrzyk et al. First investigations of the vector supercomputer SX-Aurora TSUBASA as a co-processor for database systems. In *BTW Workshops*, pages 33–50, 2019.

[20] H. Pirk et al. Voodoo - A vector algebra for portable database performance on modern hardware. *PVLDB*, 9(14):1707–1718, 2016.

[21] A. Ungethüm et al. Conflict detection-based run-length encoding - AVX-512 CD instruction set in action. In *ICDE Workshops*, pages 96–101, 2018.

Provided by Sächsische Landesb bliothek, Staats- und Universitätsbibliothek Dresden