open proceedings

# Insights into the Comparative Evaluation of Lightweight Data Compression Algorithms

Patrick Damme, Dirk Habich, Juliana Hildebrandt, Wolfgang Lehner
Database Systems Group
Technische Universität Dresden
01062 Dresden, Germany
{firstname.lastname}@tu-dresden.de

## ABSTRACT

Lightweight data compression is frequently applied in in-memory database systems to tackle the growing gap between processor speed and main memory bandwidth. In recent years, the number of available compression algorithms has grown considerably. Since the correct choice of one of these algorithms requires understanding of their performance behavior, we systematically evaluated several state-of-the-art compression algorithms on a multitude of different data characteristics. In this demonstration, the attendee will learn our findings in an interactive tour through our obtained measurements. The most important insight is that there is no single-best algorithm, but that the choice depends on the data characteristics and is non-trivial.

## 1. INTRODUCTION

The continuous growth of data volumes is a major challenge for the efficient data processing. With the growing capacity of the main memory, efficient analytical data processing becomes possible [6]. However, the gap between computing power of the CPUs and main memory bandwidth continuously increases, which is now the main bottleneck for an efficient data processing. To overcome this bottleneck, data compression plays a crucial role [1, 11]. Aside from reducing the amount of data, compressed data offers several advantages such as less time spent on load and store instructions, a better utilization of the cache hierarchy, and less misses in the translation lookaside buffer.

This compression solution is heavily exploited in modern in-memory column stores for efficient query processing [1, 11]. Here, relational data is maintained using the *decomposition storage model* [3]. That is, an $n$-attribute relation is replaced by $n$ binary relations, each consisting of one attribute and a surrogate indicating the record identity. Since the latter contains only *virtual* ids, it is not stored explicitly. Thus, each attribute is stored separately *as a sequence of values*. For the lossless compression of sequences of values (in particular integer values), a large variety of lightweight algorithms has been developed [1, 2, 7, 8, 9, 10, 11][1]. In contrast to heavyweight algorithms, lightweight algorithms achieve comparable or even better compression rates. Moreover, the computational effort for the (de)compression is lower than for heavyweight algorithms. To achieve these unique properties, each lightweight compression *algorithm* employs one or more basic compression *techniques* such as frame-of-reference [11] or null suppression [1], which allow the appropriate utilization of contextual knowledge like value distribution, sorting, or data locality.

In recent years, the efficient *vectorized* implementation of these lightweight compression algorithms using SIMD (Single Instruction Multiple Data) instructions has attracted a lot of attention [7, 8, 9, 10], since it further reduces the computational effort. To better understand these vectorized lightweight compression algorithms and to be able to select a suitable algorithm for a given data set, the behavior of the algorithms regarding different data characteristics has to be known. In particular, the behavior in terms of *performance* (compression, decompression and processing) and *compression rate* is of interest. Therefore, we have done an experimental survey of a broad range of algorithms with different data characteristics in a systematic way. We used a multitude of synthetic data sets as well as two commonly used real data sets. While we have already published *selected* results of our exhaustive evaluation in [5], this demonstration makes use of our *entire* corpus of measurements, which we obtained from an even larger collection of data characteristics than we could discuss in [5]. More precisely, the goals of this demonstration are the following:

1. We present our experimental methodology. This includes our selection of data characteristics and algorithms as well as our benchmark framework [4].
2. We explain the employed implementations of the compression algorithms and thus provide background knowledge for understanding the empirical results.
3. We explore various visualizations of the results of our systematic evaluation using an interactive web-interface.
4. Finally, we provide detailed insights into the behavior of the considered lightweight compression algorithms depending on the properties of the uncompressed data.

The remainder of the paper is organized as follows: In Section 2, we provide some important background knowledge on the area of lightweight data compression. An overview of

[1]Without claim of completeness.

our underlying systematic evaluation is given in Section 3. Finally, Section 4 describes what attendees will experience in our demonstration.

# 2. LIGHTWEIGHT DATA COMPRESSION

This section gives an overview of the basic concepts of lightweight data compression. We distinguish between compression *techniques* and compression *algorithms*, whereby each algorithm implements one or more techniques.

## 2.1 Techniques

There are five basic lightweight techniques to compress a sequence of values: frame-of-reference (FOR) [11], delta coding (DELTA) [7], dictionary compression (DICT) [1, 11], run-length encoding (RLE) [1], and null suppression (NS) [1]. FOR and DELTA represent each value as the difference to either a certain given reference value (FOR) or to its predecessor value (DELTA). DICT replaces each value by its unique key in a dictionary. The objective of these three well-known techniques is to represent the original data as a sequence of small integers, which is then suited for actual compression using the NS technique. NS is the most studied lightweight compression technique. Its basic idea is the omission of leading zeros in the bit representation of small integers. Finally, RLE tackles uninterrupted sequences of occurrences of the same value, so called *runs*. Each run is represented by its value and length. Hence, the compressed data is a sequence of such pairs.

Generally, these five techniques address different data levels. While FOR, DELTA, DICT, and RLE consider the *logical* data level, NS addresses the *physical* level of bits or bytes. This explains why lightweight data compression algorithms are always composed of one or more of these techniques. The techniques can be further divided into two groups depending on how the input values are mapped to output values. FOR, DELTA, and DICT map each input value to exactly one integer as output value (*1:1 mapping*). The objective of these three techniques is to achieve smaller numbers which can be better compressed on the bit level. In RLE, not every input value is necessarily mapped to an encoded output value, because a successive subsequence of equal values is encoded in the output as a pair of run value and run length (*N:1 mapping*). In this case, a compression is already done at the logical level. The NS technique is either a 1:1 or an N:1 mapping depending on the implementation.

## 2.2 Algorithms

The genericity of these techniques is the foundation to tailor the algorithms to different data characteristics. Therefore, a lightweight data compression algorithm can be described as a cascade of one or more of these basic techniques. On the level of the algorithms, the NS technique has been studied most extensively. There is a very large number of specific algorithms showing the diversity of the implementations for a single technique. The pure NS algorithms can be divided into the following classes [10]: (i) bit-aligned, (ii) byte-aligned, and (iii) word-aligned.[2] While bit-aligned NS algorithms try to compress an integer using a minimal number of *bits*, byte-aligned NS algorithms compress an integer with a minimal number of *bytes* (1:1 mapping). The word-aligned NS algorithms encode as many integers as possible into 32-bit or 64-bit words (N:1 mapping).

The logical-level techniques have not been considered to such an extent as the NS technique *on the algorithm level*. In most cases, they have been investigated in connection with the NS technique. For instance, PFOR-based algorithms implement the FOR technique in combination with a bit-aligned NS algorithm [11]. These algorithms usually subdivide the input in subsequences of a fixed length and calculate two parameters per subsequence: a reference value for the FOR technique and a common bit width for NS. Each subsequence is encoded using their specific parameters, thereby the parameters are data-dependently derived. The values that cannot be encoded with the given bit width are stored separately with a greater bit width.

# 3. SYSTEMATIC EVALUATION

The effective employment of these lightweight compression algorithms requires a thorough understanding of their behavior in terms of performance and compression rate. Therefore, we have conducted an extensive experimental evaluation of several lightweight compression algorithms on a multitude of different data characteristics. Furthermore, we have already published some of the results in [5]. In this section, we present the key facts about our systematic evaluation, which is the basis of the demonstration. Besides the considered algorithms and data characteristics, we also provide details on our experimental setup.

## 3.1 Considered Algorithms

Our selection of algorithms follows two principal goals: Firstly, all five techniques of lightweight data compression should be represented. Secondly, the implementations should reflect the state-of-the-art in terms of efficiency.

Regarding efficiency, the use of SIMD (Single Instruction Multiple Data) instruction set extensions such as Intel's SSE and AVX plays a crucial role. These allow the application of one operation to multiple elements of so-called vector registers at once. The available operations include parallel arithmetic, logical, and shift operations as well as permutations. These are highly relevant to lightweight compression algorithms. In fact, the main focus of recent research [7, 8, 9, 10] in this field has been the employment of SIMD instructions to speed up (de)compression. Consequently, most of the algorithms we evaluated make use of SIMD extensions.

Regarding the techniques, we consider both, algorithms implementing a single technique and cascades of one logical-level and one physical-level technique. Since implementations of the logical-level techniques are hardly available in isolation, i.e., without the combination with NS, we use our own *vectorized* reimplementions of RLE, DELTA, and FOR, and a *sequential* reimplementation of DICT. Concerning the physical-level technique NS, however, there are several publicly available high-quality implementations, e.g., the FastPFOR-library by Lemire et al.[3]. We used such available implementations whenever possible and reimplemented only the recently introduced algorithm SIMD-GroupSimple [10], since we could not find an implementation of it. Table 1 gives an overview of the NS algorithms in our systematic evaluation. Note that all three classes of NS are represented in this selection. Due to space limitations, we cannot elab-

[2] [10] also defines a *frame-based* class, which we omit, as the representatives we consider also match the *bit-aligned* class.

[3] https://github.com/lemire/FastPFOR

| Class | Algorithm | Ref. | Code origin | SIMD |
|---|---|---|---|---|
| bit-aligned | 4-Gamma | [9] | Schlegel et al. | yes |
| | SIMD-BP128 | [7] | FastPFOR-lib[3] | yes |
| | SIMD-FastPFOR | [7] | FastPFOR-lib[3] | yes |
| byte-aligned | 4-Wise NS | [9] | Schlegel et al. | yes |
| | Masked-VByte | [8] | FastPFOR-lib[3] | no/yes |
| word-aligned | Simple-8b | [2] | FastPFOR-lib[3] | no |
| | SIMD-GroupSimple | [10] | our own code | yes |

**Table 1: The considered NS algorithms.**

orate further on these algorithms. Instead, we recommend to read [5], which contains high-level descriptions of these.

To enable the *systematic* investigation of combinations of logical-level and physical-level algorithms, we implemented a generic cascade algorithm, which can be specialized for *any* pair of compression algorithms. This cascade algorithm partitions the uncompressed data into blocks of a certain size and does the following for each block: First, it applies the logical-level algorithm to the uncompressed block storing the result to a small intermediate buffer. Second, it applies the physical-level algorithm to that intermediate buffer and appends the result to the output. The decompression works the opposite way. We choose the block size such that it fits into the L1 data cache in order to achieve high performance.

To sum up, we investigated 4 logical-level algorithms, 7 physical level algorithms, and $4 \times 7 = 28$ cascades, yielding a total of 39 algorithms. For each of these algorithms, we consider the compression and decompression part. Additionally, we implemented a summation of the compressed data for each algorithm as an example of data processing.

## 3.2 Considered Data Sets

We made extensive use of *synthetic* data, since it allows us to carefully vary all relevant data properties. More precisely, we experimented with various combinations of the total number of data elements, the number of distinct data elements, the distribution of the data elements, the distribution of run lengths, and the sort order. We employed random distributions which are frequently encountered in practice, such as uniform, normal, and zipf. Additionally we introduced different amounts of outliers. For each data set, we vary one of these properties, while the others are fixed. That way, we can easily observe the impact of the varied property. To give an example, one of our data sets consists of 100 M uncompressed 32-bit integers, 90% of which follow a normal distribution with a small mean, while 10% are normally distributed outliers for which we vary the mean.

Additionally, we employed two *real* data sets which are commonly used in the literature on lightweight compression: the postings lists of the GOV2 and ClueWeb09b document collections.

## 3.3 Experimental Setup

All algorithms are implemented in C/C++ and we compiled them with `g++` 4.8 using the optimization flag `-O3`. Our Ubuntu 14.04 machine was equipped with an Intel Core i7-4710MQ (Haswell) processor with 4 physical and 8 logical cores running at 2.5 GHz. The L1 data, L2, and L3 caches have a capacity of 32 KB, 256 KB and 6 MB, respectively. We use only one core at any time of our evaluation to avoid competition for the shared L3 cache. The capacity of the DDR3 main memory was 16 GB.

All experiments happened entirely in main memory. The disk was never accessed during the time measurements. We conducted the evaluation using our benchmark framework [4]. The synthetic data was generated by our data generator once per configuration of the data properties. During the executions, the runtimes and the compression rates were measured. To achieve reliable measurements, we emptied the cache before each algorithm execution (by copying an array much larger than the L3 cache) and repeated all time measurements 12 times, whereby we used the wallclock-time.

# 4. DEMONSTRATION SCENARIO

For our demonstration we will use an interactive web-interface (Fig. 1a). This interface is based on `jupyter`[4], a tool widely used for interactive scientific data processing. While we are able to present *how* we conducted our systematic evaluation using our benchmark framework [4], the main focus of the demonstration will be the exploration of the collection of measurements.

Our demonstration offers several opportunities for involving the attendee. He or she can select the algorithms to be compared as well as the data characteristics (Fig. 1a). Optionally, the attendee can define a trade-off between the possible optimization goals of lightweight compression. For instance, there are scenarios in which the decompression speed is most relevant, while the compression speed is not of interest, or in which the compression rate is more important than the speeds. Defining such a trade-off can help to determine the best algorithm for a given use case.

Regarding the available data sets, we already have a very large collection of measurements, which we obtained in our systematic evaluation as mentioned in Section 3.2. These reflect a multitude of combinations of relevant data properties. Nevertheless, if the attendee wishes so, he or she can also define a configuration of data characteristics, we have not considered so far. In this case, we would simply run the evaluation on the fly.

While it is possible to investigate any of the considered compression algorithms on any data in an interactive and spontaneous way, we would like to highlight the following prepared scenarios:

- *Impact of the data distribution on the physical-level algorithms* (Fig. 1b). In this scenario, the attendee will learn about the general behavior of null suppression algorithms depending on their class. Furthermore, he or she will find out, how different data distributions influence this behavior additionally.
- *Impact of the logical-level algorithms on the data characteristics* (Fig. 1c). Here, the attendee can observe how the application of purely logical-level algorithms such as RLE and FOR changes the properties of the underlying data. For this purpose, we employ, e.g., visualizations of the data distributions. We will discuss with the attendee, in how far these changed properties are suited for the following application of a physical-level algorithm.
- *Cascades of logical-level and physical-level algorithms* (Fig. 1d). Finally, the attendee will learn that the combination of logical-level and physical-level algorithms

[4]http://www.jupyter.org

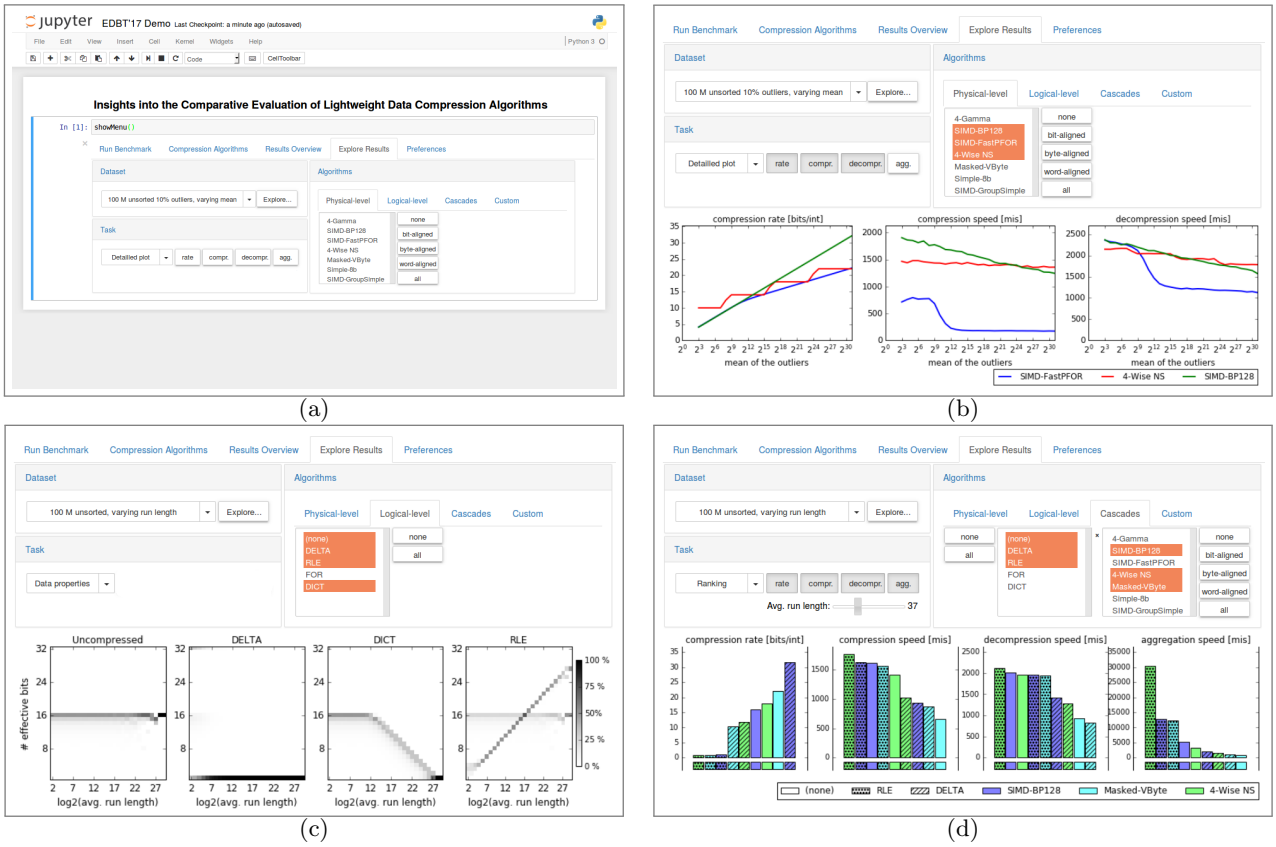


**Figure 1: Screenshots of our demonstration web-interface.**

can yield significant improvements in terms of speed or compression rate, but not necessarily both. Defining a trade-off can help to make a final decision. Moreover, depending on the data, not all combinations are beneficial. The attendee will understand that even if the logical-level technique is fixed, the choice of the NS algorithm can make a significant difference.

By the end of the demonstration, the attendee will appreciate that there is no single-best lightweight compression algorithm, but that the choice depends on the data characteristics as well as the optimization goal and is non-trivial. At this point, the results of our systematic evaluation can help to select the best algorithm for a given data set.

## Acknowledgments

This work was partly funded by the German Research Foundation (DFG) in the context of the project "Lightweight Compression Techniques for the Optimization of Complex Database Queries" (LE-1416/26-1).

## 5. REFERENCES

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

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

[3] G. P. Copeland and S. N. Khoshafian. A decomposition storage model. *SIGMOD Rec.*, 14(4), 1985.

[4] P. Damme, D. Habich, and W. Lehner. A benchmark framework for data compression techniques. In *TPCTC*, 2015.

[5] P. Damme, D. Habich, and W. Lehner. Lightweight data compression algorithms: An experimental survey. In *EDBT*, 2017.

[6] T. Kissinger, T. Kiefer, B. Schlegel, D. Habich, D. Molka, and W. Lehner. ERIS: A numa-aware in-memory storage engine for analytical workloads. In *ADMS*, 2014.

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

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

[9] B. Schlegel, R. Gemulla, and W. Lehner. Fast integer compression using SIMD instructions. In *DaMoN*, 2010.

[10] 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), 2015.

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