Lightweight Data Compression Algorithms: An Experimental Survey

Experiments and Analyses

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 algorithms are frequently applied in in-memory database systems to tackle the growing gap between processor speed and main memory bandwidth. In recent years, the vectorization of basic techniques such as delta coding and null suppression has considerably enlarged the corpus of available algorithms. As a result, today there is a large number of algorithms to choose from, while different algorithms are tailored to different data characteristics. However, a comparative evaluation of these algorithms under different data characteristics has never been sufficiently conducted in the literature. To close this gap, we conducted an exhaustive experimental survey by evaluating several state-of-the-art compression algorithms as well as cascades of basic techniques. We systematically investigated the influence of the data properties on the performance and the compression rates. The evaluated algorithms are based on publicly available implementations as well as our own vectorized reimplementations. We summarize our experimental findings leading to several new insights and to the conclusion, that there is no single-best algorithm.

1. INTRODUCTION

The continuous growth of data volumes is a major challenge for the efficient data processing. This applies not only to database systems [1, 5] but also to other areas, such as information retrieval [3, 18] or machine learning [8]. With the growing capacity of the main memory, efficient analytical data processing becomes possible [4, 11]. 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, 22]. 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, 22]. Here, relational data is maintained using the *decomposition storage model* [6]. 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, 3, 9, 12, 15, 16, 17, 18, 22][1]. In contrast to heavyweight algorithms like arithmetic coding [19], Huffman [10], or Lempel Ziv [21], 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 [9, 22] or null suppression [1, 15], 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 [12, 14, 16, 18, 20], 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. In the literature, there are two papers with a considerable evaluation part. First, Adabi et al. [1] evaluated a small number of *unvectorized* algorithms on different data characteristics, but they neither considered a rich set of data distributions nor the explicit combination of different compression techniques. Second, Lemire et al. [12] already evaluated *vectorized* lightweight data compression algorithms, but considered only null suppression with and without differential coding. Furthermore, their focus is on postings lists from the IR domain, which narrows the considered data characteristics. Hence, an exhaustive comparative evaluation as a foundation has been never sufficiently conducted. To overcome this issue, we have

[1] Without claim of completeness.

Series ISSN: 2367-2005

10.5441/002/edbt.2017.08

done an experimental survey of a broad range of algorithms with different data characteristics in a systematic way. In our evaluation, we used a set of synthetic data sets as well as one commonly used real data set. Our main findings can be summarized as follows:

1. Performance and compression rate of the algorithms vary greatly depending on the data properties. Even algorithms that are based on the same techniques, show a very different behavior.
2. By combining various basic techniques, the compression rate can be improved significantly. The performance may rise or fall depending on the combination.
3. There is no single-best lightweight algorithm, but the decision depends on the data properties. In order to select an appropriate algorithm, a compromise between performance and compression rate must be defined.

The remainder of the paper is organized as follows: In Section 2, we present more details about the area of lightweight data compression and introduce our evaluated algorithms. The implementation aspects are described in Section 3, while Section 4 covers our evaluation setup. Selected results of our experimental survey are presented in Section 5 and Section 6. Finally, we conclude the paper in Section 7.

2. PREREQUISITES

The focus of our experimental survey is the large corpus of *lossless lightweight integer data compression algorithms* which are heavily used in modern in-memory column stores [1, 22]. To better understand the algorithm corpus, this section briefly summarizes the basic concepts and introduces the algorithms which are used in our survey.

2.1 Lightweight Data Compression

First of all, we have to distinguish between *techniques* and *algorithms*, thereby each algorithm implements one or more of these techniques.

Techniques. There are five basic lightweight techniques to compress a sequence of values: frame-of-reference (FOR) [9, 22], delta coding (DELTA) [12, 15], dictionary compression (DICT) [1, 22], run-length encoding (RLE) [1, 15], and null suppression (NS) [1, 15]. 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. In the following, we also denote the techniques from the logical level as preprocessing techniques for the physical compression with NS. These 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 preprocessing 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.

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 lightweight data compression 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 [20]: (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 integer values as possible into 32-bit or 64-bit words (N:1 mapping). The NS algorithms also differ in their data layout. We distinguish between *horizontal* and *vertical* layout. In the horizontal layout, the compressed representation of subsequent values is situated in subsequent memory locations. In the vertical layout, each compressed representation is stored in a separate memory word.

The logical-level techniques have not been considered to such an extent as the NS technique *on the algorithm level.* In most cases, the preprocessing steps 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 [22]. 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.

2.2 Considered Algorithms

We consider all five basic lightweight techniques in detail. Regarding the selected algorithms, we investigate both, implementations of a single technique as well as cascades of one logical-level and one physical-level technique. We decided to reimplement the logical-level techniques on our own (see Section 3) in order to be able to freely combine them with all seven considered NS algorithms (see Table 1). In the following, we briefly sketch each considered NS algorithm.

2.2.1 Bit-Aligned NS Algorithms

4-Gamma Coding [16] processes four input values (data elements) at a time. All four values are stored in the vertical storage layout using the number of bits required for the

[2] [20] also defines a *frame-based* class, which we omit, as the representatives we consider also match the *bit-aligned* class.

largest of them. The unary representation of the bit width is stored in a separate memory area for decompression.

SIMD-BP128 [12] processes data in blocks of 128 integers at a time. All 128 integers in the block are stored in the vertical layout using the number of bits required for the largest of them. The used bit width is stored in a single byte, whereby 16 of these bit widths are followed by 16 compressed blocks.

SIMD-FastPFOR [12] is a variant of the original PFOR algorithm [22], whose idea is to classify all data elements as either regular coded values or exceptions depending on if they can be represented with a certain bit width. This bit width is chosen such that the overall compression rate becomes optimal. All data elements are packed with the chosen bit width using the vertical layout. The exceptions require a special treatment, since that number of bits does not suffice for them. In SIMD-FastPFOR the exceptions are stored in additional packed arrays. The overall input is subdivided into pages which are further subdivided into blocks of 128 integers. SIMD-FastPFOR stores the exceptions at the page level and uses an individual bit width for each block.

2.2.2 *Byte-Aligned NS Algorithms*

4-Wise Null Suppression [16] compresses integers by omitting leading zero bytes. For each 32-bit integer, between zero and three bytes might be omitted. 4-Wise NS processes four data elements at a time and combines the corresponding four 2-bit descriptors into a 1-byte mask. In the output, four masks are followed by four compressed blocks in the horizontal layout.

Masked-VByte [14] uses the same compressed representation as the VByte algorithm [12] and differs only in implementation details. It subdivides an integer into 7-bit units. Each unit that is required to represent the integer produces one byte in the output. The seven data bits are stored in the lower part of that byte, while the most significant bit is used to indicate whether or not the next byte belongs to the next data element. Subsequent compressed values are stored using the horizontal layout.

2.2.3 *Word-Aligned NS Algorithms*

Simple-8b [2] outputs one compressed block of 64 bits for a variable number of uncompressed integers. Within one block, all data elements are stored with a common bit width using the horizontal layout. The bit width is chosen such that as many subsequent input elements as possible can be stored in the compressed block. One compressed block contains 60 data bits and a 4-bit selector specifying the compression mode. There are 16 compression modes: 60 1-bit values, 30 2-bit values, 20 3-bit values, and so on. Additionally, Simple-8b has two special modes indicating that the input consisted of 120 respectively 240 zeroes.

SIMD-GroupSimple [20] processes the input in units of so-called *quads*, i.e., four values at a time. For each quad, it determines the number of bits required for the largest element. Based on the bit widths of subsequent quads, it partitions the input sequence into groups, such that as many quads as possible can be stored in *four* consecutive 32-bit words using the vertical layout. There are ten compression modes: the four consecutive 32-bit words could be filled with 4 × 32 1-bit values, 4 × 16 2-bit values, 4 × 10 3-bit values, and so on. A four bit selector represents the mode chosen for the compressed block. The selectors are stored in a different memory area than the compressed blocks.

Class	Algorithm	Layout	Code origin	SIMD
bit-aligned	4-Gamma	vert.	Schlegel et al.	yes
	SIMD-BP128	vert.	FastPFor-lib	yes
	SIMD-FastPFOR	vert.	FastPFor-lib	yes
byte-aligned	4-Wise NS	horiz.	Schlegel et al.	yes
	Masked-VByte	horiz.	FastPFor-lib	n/y
word-aligned	Simple-8b	horiz.	FastPFor-lib	no
	SIMD-GroupSimple	vert.	our own code	yes

Table 1: The considered NS algorithms.

3. IMPLEMENTATION ASPECTS

As already mentioned, we reimplemented all four logical-level techniques in C++, i.e., DELTA, DICT, FOR, and RLE. Regarding the physical-level, several high-quality open-source implementations of NS are available. We used these existing implementations whenever possible and reimplemented only one of them. Table 1 summarizes the origins of the implementations we employed. We also implemented cache-conscious generic cascades of logical-level techniques and NS. Furthermore, we implemented a decompression with aggregation for all algorithms to evaluate a processing of compressed data. In this section, we describe some of the most crucial implementation details with respect to performance.

3.1 SIMD Instruction Set Extensions

Single Instruction Multiple Data (SIMD) instruction set extensions such as Intel's SSE and AVX have been available in modern processors for several years. SIMD instructions apply 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 [12, 14, 16, 18, 20] in this field has been the employment of SIMD instructions to speed up (de)compression. Consequently, most algorithms we evaluate in this paper make use of SIMD extensions (see Table 1). Vectorized load and store instructions can be either aligned or unaligned. The former require the accessed memory addresses to be multiples of 16 bytes (SSE) and are usually faster. Although nowadays Intel's AVX2 offers 256-bit operations, we decided to restrict our evaluation to implementations using 128-bit SSE. This has two reasons: (1) Most of the techniques presented in the literature are designed for 128-bit vector operations[3] and (2) The comparison is fairer if only one width of vector registers is considered. Intel's SIMD instructions can be used in C/C++ without writing assembly code via intrinsic functions, whose names start with `_mm_`.

3.2 Physical-Level Technique: NS

In the following, we describe crucial points regarding existing implementations as well as one reimplementation.

3.2.1 *Bit-Aligned Algorithms*

We obtained the implementation of **4-Gamma Coding** directly from the authors [16], and those of **SIMD-BP128** and **SIMD-FastPFor** from the FastPFor-library [13]. All

[3] Thereby, a transition to 256-bit operations is not always trivial and could be subject to future research.

three implementations use vectorized shift and mask operations. SIMD-BP128 and SIMD-FastPFor use a dedicated optimized packing and unpacking routine for each of the 32 possible bit widths. It is worth mentioning, that – while the original PFOR algorithm is a combination of the FOR and the NS technique – SIMD-FastPFOR, despite its name, does not include the FOR technique, but only the NS technique.

3.2.2 *Byte-Aligned Algorithms*

Regarding **4-Wise Null Suppression**, we use the original implementation by Schlegel et al. [16]. It implements the horizontal packing of the uncompressed values using a vectorized byte permutation. The 16-byte permutation masks required for this are built once in advance and looked up from a table during the compression. This table is indexed with the 1-byte compression masks, thus there are 256 permutation masks in total. The decompression works by using the inverse permutation masks.

Masked-VByte vectorizes the decompression of the compressed format of the original VByte algorithm. The implementation we use is available in the FastPFor-library [13] and is based on code by the original authors. The crucial point of the vectorization is the execution of a SIMD byte permutation in order to reinsert the leading zero-bytes removed by the compression. After 16 bytes of compressed data have been loaded into a vector register, the most significant bits of all bytes are extracted using a SIMD instruction. The lower 12 bits of this 16-bit mask are used as a key to lookup the required permutation mask in a table. After the permutation, the original 7-bit units need to be stitched together, which is done using vectorized shift and mask operations. Masked-VByte also has an optimization for the case of 12 compressed 1-byte integers.

3.2.3 *Word-Aligned Algorithms*

We use the implementation of **Simple-8b** available in the FastPFor-library [13], which is a purely sequential implementation. It uses a dedicated sequential packing routine for each of the possible selectors.

We reimplemented **SIMD-GroupSimple** based on the description in the original paper, since we could not find an available implementation. We employed the two optimizations discussed by the original authors: (1) We calculate the pseudo-quad max values instead of the quad max values to reduce the number of branch instructions. (2) We use one dedicated and vectorized packing routine for each selector, whereby the correct one is chosen by a switch-statement.

The original compression algorithm processes the input data in *three* runs: The first run scans the entire input and materializes the pseudo-quad max array in main memory. The size of this array is one quarter of the input data size. The second run scans the pseudo-quad max array and materializes the selectors array. The third run iterates over the selectors array and calls the respective packing routine to do the actual compression. This procedure results in a suboptimal cache utilization, since at the end of each run, the data it started with has already been evicted from the caches. Thus, reaccessing it in the next run becomes expensive.

In order to overcome this issue, we enhanced the compression part of the algorithm with one more optimization, which was not presented in the original paper: Our reimplementation stores the pseudo-quad max values in a ring buffer of a small constant size (32 32-bit integers) instead of an array proportional to the input size. This is based on the observation that the decision for the next selector can never require more than 32 pseudo-quad max values, since at most 4×32 (1-bit) integers can be packed into four 32-bit words. Due to its small size (128 bytes), the ring buffer fits into the L1 data cache and can thus be accessed at top-speed. Our modified compression algorithm repeats the following steps until the end of the input is reached (in the beginning, the ring buffer is empty): (1) Fill the ring buffer by calculating the next up to 32 pseudo-quad max values. This reads up to $4 \times 32 = 128$ uncompressed integers. (2) Run the original subroutine for determining the next selector *on the ring buffer*. (3) Store the obtained selector to the selectors section in the output. (4) Compress the next block using the subroutine belonging to the selector. This will typically reread the uncompressed data touched in Step 1. Note that this data is very likely to still reside in the L1 cache, since only a few bytes of memory have been touched in between. (5) Increase the position in the ring buffer by the number of input quads compressed in the previous step. We observed that using this additional optimization, the compression part of our reimplementation is always faster than without it. Note, that this optimization does not affect the compressed output in any way.

3.3 Logical-Level Techniques

As previously mentioned, logical-level techniques are usually combined with NS in existing algorithms and are thus hardly available in isolation. In order to be able to freely combine *any* logical-level technique with *any* NS algorithm, we reimplemented all four logical-level compression techniques as stand-alone algorithms. Thereby, an important goal is the vectorization of those algorithms.

3.3.1 *Vectorized DELTA*

Our implementation of DELTA represents each input element as the difference to its fourth predecessor. This allows for an easy vectorization by processing four integers at a time. The first four elements are always copied from the input to the output. During the compression, the next four differences are calculated at once using `_mm_sub_epi32()`. The decompression reverses this by employing `_mm_add_epi32()`. This implementation follows the description in [12] with the difference that we do not overwrite the input data, because we still need it as the input for the other algorithms.

3.3.2 *Sequential DICT*

Our implementation of DICT is a purely sequential single-pass algorithm employing a static dictionary, which is built on the uncompressed data before the (de)compression takes place. Thus, building the dictionary is not included in our time measurements and the dictionary itself is not included in the compressed representation. This represents the case of a domain-specific dictionary which is known in advance. The compression uses a C++-STL `unordered_map` to map values to their keys, whereas the decompression uses the key as the index of a `vector` to look up the corresponding value.

3.3.3 *Vectorized FOR*

We implemented the compression of FOR as a vectorized two-pass algorithm. The first pass iterates over the input and determines the reference value, i.e., the minimum using `_mm_min_epu32()`. This minimum is then copied into all four elements of one vector register. The second pass iterates over the input again and subtracts this vector register from four input elements at a time using `_mm_sub_epi32()`. In the end, the reference value is appended to the output. The decompression adds this reference value to four data elements at a time using `_mm_add_epi32()`.

3.3.4 *Vectorized RLE*

Our implementation of RLE also utilizes SIMD instructions. The compression part is based on parallel comparisons. It repeats the following steps until the end of the input is reached: (1) One 128-bit vector register is loaded with four copies of the current input element. (2) The next four input elements are loaded. (3) The intrinsic `_mm_cmpeq_epi32()` is employed for a parallel comparison. The result is stored in a vector register. (4) We obtain a 4-bit comparison mask using `_mm_movemask_ps()`. Each bit in the mask indicates the (non-)equality of two corresponding vector elements. The number of trailing one-bits in this mask is the number of elements for which the run continues. If this number is 4, then we have not seen the run's end yet, and continue at step 2. Otherwise, we have reached the run's end and append the run value and run length to the output and continue with step 1 at the next element after the run's end.

The decompression executes the following until the entire input has been consumed: (1) Load the next pair of run value and run length. (2) Load one vector register with four copies of the run value. (3) Store the contents of that register to memory as often as required to match the run length.

3.4 Cascades of Logical-Level and Physical-Level Techniques

The challenge of implementing cascades, i.e., combinations of logical-level and physical-level techniques, is the high implementation effort due to the high number of possible combinations. To address this problem, we implemented a cache-conscious cascade which is generic w.r.t. the employed algorithms. That is, it can be instantiated for *any* two algorithms, without further implementation effort. It takes three parameters: a logical-level algorithm L, a physical-level algorithm P, and an (uncompressed) block size bs_u.

The output consists of compressed blocks, each of which starts with its size as a 32-bit integer followed by 12 bytes of padding to achieve the 16-byte alignment required by SSE instructions. The body of the block contains the compressed data possibly followed by additional padding bytes.

The compression procedure repeats the following steps until the end of the input is reached: (1) Skip 16 bytes in the output buffer. (2) Apply the compression of L to the next bs_u elements in the input. Store the result in an intermediate buffer. (3) Apply the compression of P to that buffer and store the result to the output buffer. (4) Store the size bs_c of the compressed block to the bytes skipped in Step 1. (5) Skip some bytes after the compressed block, if it is necessary to achieve 16-byte alignment.

The decompression is the reverse procedure repeatedly executing the following steps: (1) Read the size bs_c of the current compressed block and skip the padding. (2) Apply the decompression of P to the next bs_c bytes in the input. Store the result to an intermediate buffer. (3) Decompress the contents of that buffer using L and append the result to the output. (4) Skip the padding in the input, if necessary.

The intermediate buffer is reused for all blocks. Its size is in the order of magnitude of bs_u (we chose $4KiB + 2 \times bs_u$ as a pessimistic estimation). This algorithm is cache-conscious, if bs_u is chosen to fit the Lx cache, since then, the data read by the second algorithm is likely to still reside in that cache.

3.5 Decompression with Aggregation

We also modified the decompressions of both, our own reimplementations and existing implementations, such that they sum up the decompressed data instead of writing it to memory. The usual case for the vectorized algorithms is that four decompressed 32-bit integers reside in a vector register before they are stored to memory using `_mm_store_si128()`. We replaced these store instructions by vectorized additions. However, since the sum might require more than 32 bits, we first distribute the four 32-bit elements to the four 64-bit elements of *two* 128-bit registers using `_mm_unpacklo_epi32()` and `_mm_unpackhi_epi32()` and add both to *two* 64-bit running sums (which are added in the very end) by applying `_mm_add_epi64()`. In the case of RLE, we add the product of the run length and the run value to the running sum.

4. EVALUATION SETUP

In this section, we describe our overall evaluation setup. All algorithms are implemented in C/C++ and we compiled them with `g++` 4.8 using the optimization flag `-O3`. As the operating system we used Ubuntu 14.04. All experiments have been executed on the same hardware machine in order to be able to compare the results. The 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. We are able to copy data using `memcpy()` at a rate of 6.15 GiB/s or 1,650 mis (million integers per second).

All experiments happened entirely in main memory. The disk was never accessed during the time measurements. The whole evaluation is performed using our benchmark framework [7]. The synthetic data generation was performed by our data generator once per configuration of data properties. The data properties were recorded and the algorithms were repeatedly performed on the generated data. During the executions, the runtimes and the compression rates were measured. Furthermore, we emptied the cache before each algorithm execution by copying a 12-MB array (twice as large as the L3 cache) using a loop operation.

All time measurements were carried out by means of the wallclock-time (C++-STL `high_resolution_clock`) and were repeated 12 times to receive stable values, thereby we only report average values. The time measurements include:

Compression: Loading uncompressed data from main memory, applying the compression algorithm, storing the compressed data to main memory

Decompression: Loading compressed data from main memory, applying the decompression algorithm, storing the uncompressed data to main memory

Decompression & Aggregation: Loading compressed data from main memory, applying the decompression and summation, storing 8 bytes in total to main memory

5. EXPERIMENTS ON SYNTHETIC DATA

In this section, we present selected results of our experi-

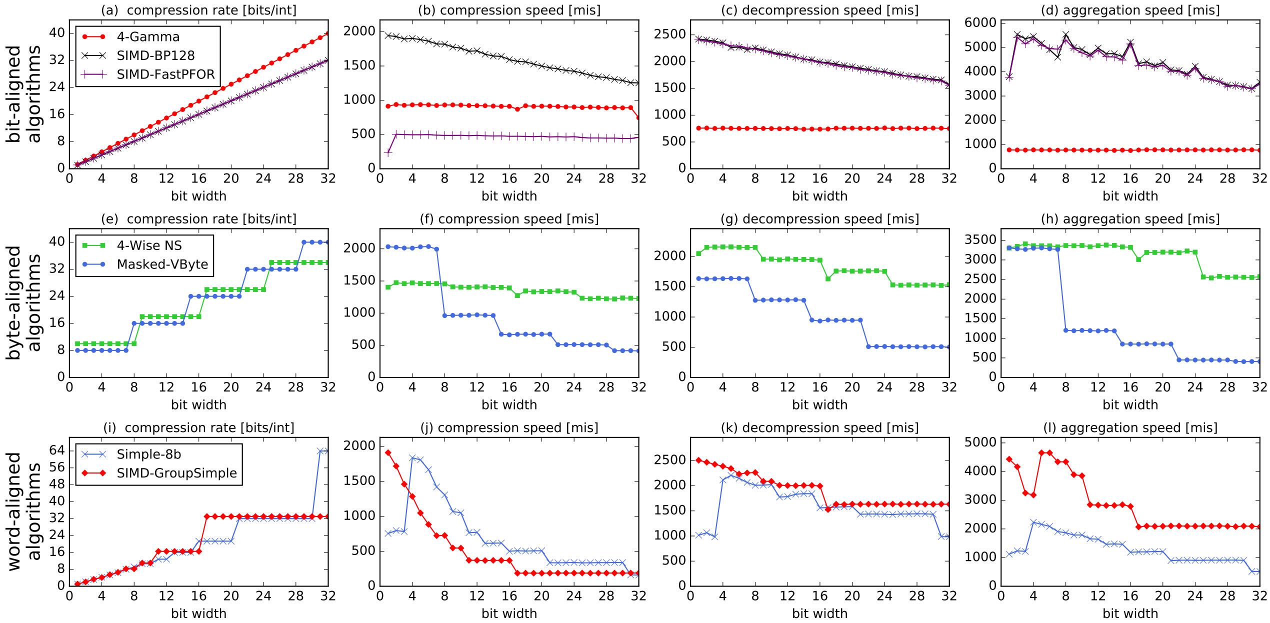

Figure 1: The general behavior of the three classes of NS algorithms.

mental survey. We generate synthetic data sets in order to be able to control the data properties in a systematic way. All uncompressed arrays contain 100 million 32-bit integers, i.e., 400 MB. Thus, only a small portion of the uncompressed data fits into the L3 cache. We report speeds in *million integers per second (mis)* and compression rates in *bits per integer (bits/int)*. We begin with the evaluation of pure NS algorithms in Section 5.1. After that, we investigate pure logical-level algorithms in Section 5.2. In Section 5.3, we evaluate cascades of logical-level techniques and NS. Finally, in Section 5.4 we present the conclusions we draw from our evaluation on synthetic data.

5.1 Null Suppression Algorithms

We start by identifying the characteristics of the three classes of NS algorithms. After that, we compare five selected NS algorithms in more detail.

5.1.1 Classes of NS Algorithms

We generate 32 unsorted datasets, such that all data elements in the i-th dataset have exactly i effective bits, i.e., the value range is $[0, 1]$ for $i = 1$ and $[2^{i-1}, 2^i)$ for $i = 2, \ldots, 32$. Within these ranges, the values are uniformly distributed.

Figure 1 (a-d) show the results for the considered **bit-aligned algorithms**. These have the finest possible compression granularity and thus can perfectly adapt to any bit width. Consequently, the compression rate is a linear function of the bit width. The speeds of compression, decompression, and aggregation follow the same linear trend. Nevertheless, there are differences between the algorithms. Since SIMD-BP128 and SIMD-FastPFOR store less meta information than 4-Gamma, they achieve better compression rates. They are also better regarding the decompression and aggregation speed. However, in terms of compression speed, SIMD-FastPFOR is by far the slowest, while SIMD-BP128 still shows very good performance.

Figure 1 (e-h) present the results for the considered **byte-aligned** algorithms. These algorithms compress an integer at the granularity of units of 8 bits (4-Wise NS) or 7 bits (Masked-VByte). As a result, the curves of all four measured variables exhibit a step-shape, whereby the step width is constant and equals the unit size of the algorithm. Since the units of 4-Wise NS and Masked-VByte have different sizes, the first and second regarding compression rate change several times when increasing the bit width. Concerning the speeds, 4-Wise NS is always at least as fast as Masked-VByte, except for the compression of values with up to 7 bits, in this case Masked-VByte is significantly faster.

Finally, Fig. 1 (i-l) provide the results for the **word-aligned** algorithms. These can adapt only to certain bit widths, which are not the multiples of any unit size. For instance, SIMD-GroupSimple supports 1, 2, 3, 4, 5, 6, 8, 10, 16, and 32 bits. Hence, the measured variables show steps at these bit widths, i.e., the steps do not have a constant width per algorithm. This basic shape is especially clear for the compression rate and the compression speed, but can also be found in the decompression and aggregation speed. SIMD-GroupSimple compresses better for certain bit widths, and for others, Simple-8b does. Since Simple-8b uses 64-bit words in the output, it can still achieve a size reduction for bit widths up to 20, while SIMD-GroupSimple cannot reduce the size anymore if the bit width exceeds 16. SIMD-GroupSimple is faster for bit widths up to 3 and slower in all other situations. However, regarding decompression and aggregation, it is faster for all bit widths.

To summarize, each of the three classes exhibits its individual behavior subject to the bit width. At the same time the differences between the classes are significant.

5.1.2 Detailed Comparison of NS Algorithms

For the following experiments we pick SIMD-BP128, SIMD-FastPFOR, 4-Wise NS, Masked-VByte, and SIMD-GroupSimple and investigate their behavior in more detail. Note that all three classes of NS are represented in this selection.

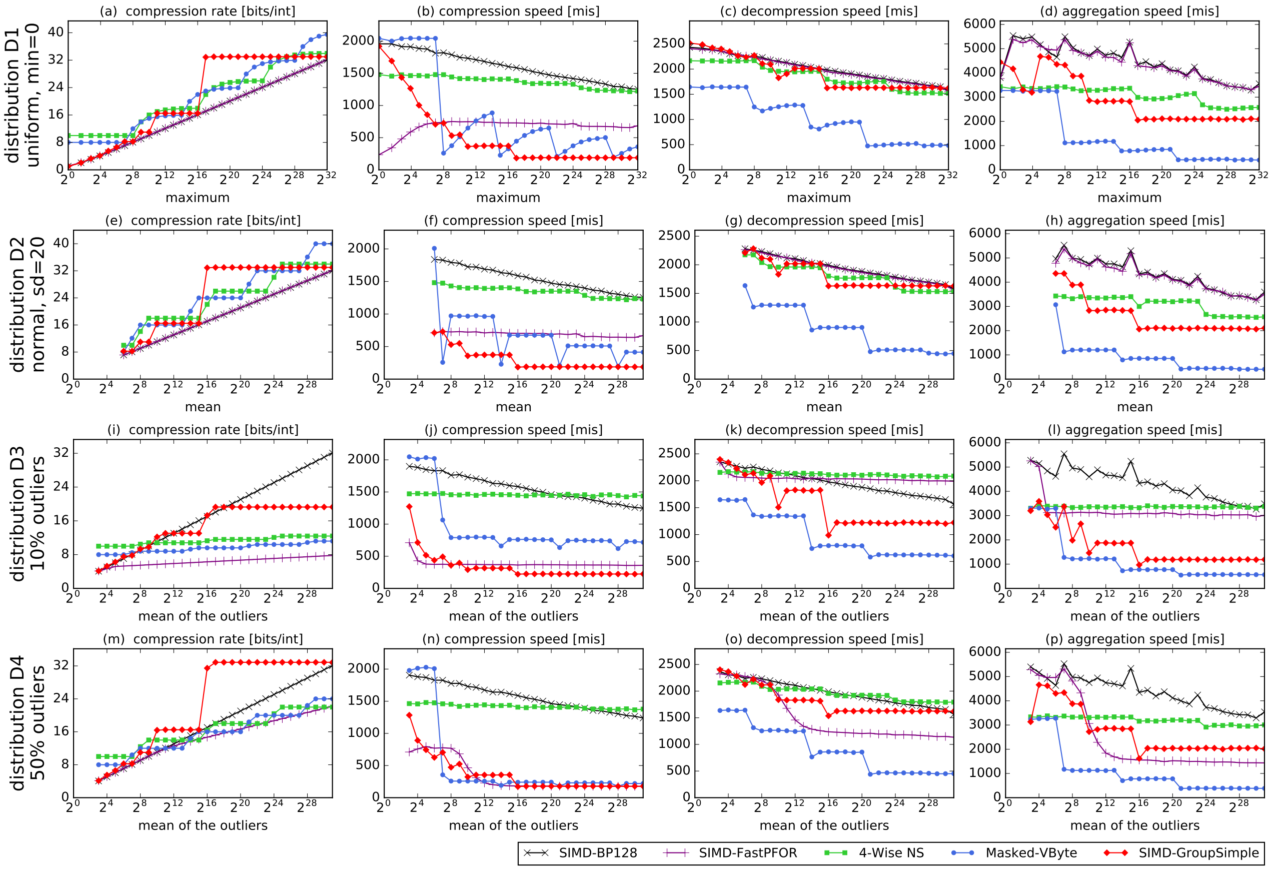

Figure 2: Comparison of NS algorithms of different classes on different data distributions.

We generate unsorted data using four distributions D1–4, whereby we vary one parameter for each of them. D1 is a uniform distribution with a min of 0 and a max varying from 0 to $2^{32} - 1$. D2 is a normal distribution with a standard deviation of 20 and a mean varying from 64 to 2^{31}. For D3, 90% of the values follow a normal distribution with a standard deviation of 2 and a mean of 8, while 10% are drawn from a normal distribution with the same standard deviation and a mean varying from 8 to 2^{31}. That is, 90% of the data elements are small integers, while 10% are increasingly large outliers. D4 is like D3, but with a ratio of 50:50. While D1–2 have a high data locality, D3–4 do not.

The results for D1 can be found in Fig. 2 (a-d). The bit-aligned algorithms SIMD-BP128 and SIMD-FastPFOR always achieve the best compression rates, since they can adapt to any bit width. Masked-VByte is the fastest compressor for small values, although it is not even vectorized. However, for larger values, SIMD-BP128 is the fastest, but comes closer to 4-Wise NS as the values grow. SIMD-GroupSimple yields the highest decompression speed for maximums up to 32. From there on SIMD-BP128 and SIMD-FastPFOR are the fastest, while SIMD-GroupSimple and 4-Wise NS come quite close to their performance, especially for the values for which they do not waste too many bits due to their coarser granularity.

For D2 (Fig. 2 (e-h)) we can make the same general observations. However, the steps in the curves of the byte-aligned algorithms become steeper, since D2 produces values with less distinct bit widths than D1.

The results of D3 (Fig. 2 (i-l)) reveal some interesting effects. Regarding the compression rate, SIMD-FastPFOR stays the winner, while SIMD-BP128 is competitive only for small outliers. For large outliers it even yields the worst compression rates of all five algorithms. This is due to the fact that SIMD-BP128 packs blocks of 128 integers with the bit width of the largest element in the block, i.e., one outlier per block affects the compression rate significantly. SIMD-FastPFOR on the other side, can handle this case very well, since it – like all variants of PFOR – is explicitly designed to tolerate outliers. The byte-aligned algorithms 4-Wise NS and Masked-VByte are worse than SIMD-FastPFOR, but still quite robust, since they choose an individual byte width for each data element and are, thus, not affected by outliers. SIMD-GroupSimple compresses better than SIMD-BP128 in most cases, since outliers lead to small input blocks, while there can still be large blocks of non-outliers. In terms of compression speed, SIMD-BP128 is still in the top-2, but it is overtaken by 4-Wise NS for large outliers. Concerning decompression speed, 4-Wise NS overtakes SIMD-BP128 when the outliers need more than 12 bits. SIMD-FastPFOR is nearly as fast as 4-Wise NS, but achieves much better compression rates. Regarding the aggregation, SIMD-BP128 is still the fastest algorithm, although SIMD-FastPFOR comes very close for small outliers and 4-Wise NS for large outliers.

D4 increases the amount of outliers to 50%. The compression rate of SIMD-BP128 does not change any more, since

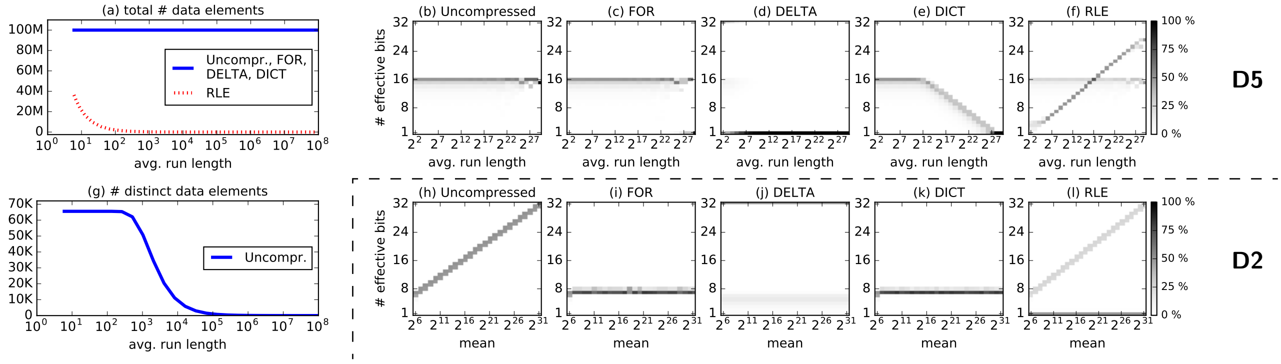

Figure 3: Logical-level techniques applied to D5 (a-g) and D2 (h-l): Data properties.[4]

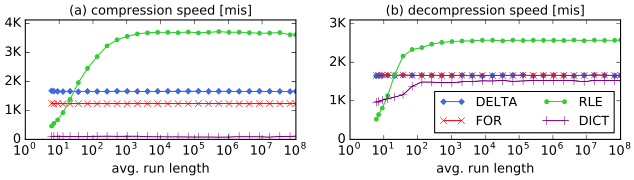

Figure 4: Logical-level techniques on D5: Speeds.

basically all blocks were affected by outliers in D3 already. However, since the other algorithms compress worse now, the trade-offs have to be reevaluated. Thanks to patched coding, SIMD-FastPFOR still is in the top-2 regarding the compression rate. However, this comes at the cost of (de)-compression and aggregation performance, which heavily decreases as the outliers grow. Encoding each value individually 4-Wise NS and Masked-VByte come very close to the compression rate of SIMD-FastPFOR and 4-Wise NS decompresses faster than SIMD-FastPFOR for large outliers.

To sum up, the best algorithm regarding compression rate or performance depends on the data distribution. Regarding one measured variable, a certain algorithm can be the best for one distribution and the worst for another distribution. Moreover, for a certain distribution the best algorithm regarding one measured variable can be the worst for another variable. In addition, there are many points of intersection between the algorithms' compression rates and speeds offering many different trade-offs.

5.2 Logical-Level Techniques

A general trend observable in Figures 1 and 2 is that all NS algorithms get worse as the data elements get larger. Logical-level techniques can be able to change the data properties in favor of NS. To illustrate this, we provide the results of the application of the four logical techniques to two *unsorted* datasets: D2, already known from the previous section, and D5, whose data elements are uniformly drawn from the range $[0, 2^{16} - 1]$ while varying the average run length.

We start with the discussion of D5. First of all, in Fig. 3 (a) we can see that the total number of data elements after the application of FOR, DELTA, and DICT is the same as in the uncompressed data (1:1 mapping), while with RLE it decreases significantly as the run length increases (N:1 mapping). This has two consequences: (1) an NS algorithm applied after RLE needs to compress less data and (2) RLE alone suffices to reduce the data size. Figure 3 (b-f)[4] show the data distributions in the uncompressed data as well as in the outputs of the logical-level techniques. Most uncompressed values have 16 or 15 effective bits. This does not change much with FOR, since the value distribution can produce values close to zero. In contrast, the output of DELTA contains nearly only values of one effective bit for long runs, since these yield long sequences of zeros. Note that there are also outliers having 32 effective bits, resulting from negative differences being represented in the two's complement. With DICT, the values start to get smaller as soon as the run length is high enough to lead to a decrease of the number of distinct values (see Fig. 3 (g)), and thus the maximum key. For RLE there are always two peaks in the distributions: one is at a bit width of 16 and corresponds to the run values and the other one is produced by the increasingly high run lengths. Note that this distribution is quite similar to D4 from the previous section. The distributions might seem to get worse for high run lengths. However, it must be kept in mind that RLE reduces the total number of data elements in those cases. Figure 4 provides the (de)compression speeds. The performance of DELTA and FOR is independent of the data characteristics, since they execute the same instructions for each group of four values. On the other side, RLE is slow for short runs, but becomes by far the fastest algorithm for long runs, since it has to write(read) less data during the (de)compression. DICT is the slowest compressor due to the expensive look ups in the map. Regarding the decompression, it is competitive to DELTA and FOR, but sensitive to the number of distinct values, which influences whether or not the dictionary fits into the Lx cache.

The distributions for D2 are visualized in Fig. 3 (h-l). Here, FOR can improve the distribution significantly, since the value range is narrow. The same applies to DICT, since consequently the number of distinct values is small. As the data is unsorted and does not have runs, about half of the values in the output of DELTA have 32 effective bits, i.e., the distributions get worse in most cases. Note that RLE doubles the number of data elements due to the lack of runs.

To sum up, logical-level techniques can significantly im-

[4] How to read Fig. 3 (b-f) and (h-l): The y-axis lists all possible numbers of effective bits a data element could have. Each vertical slice corresponds to one configuration of the data properties. The intensity encodes what portion of the data elements has how many effective bits. That is, the dark pixels show which numbers of effective bits occur most frequently in the dataset.

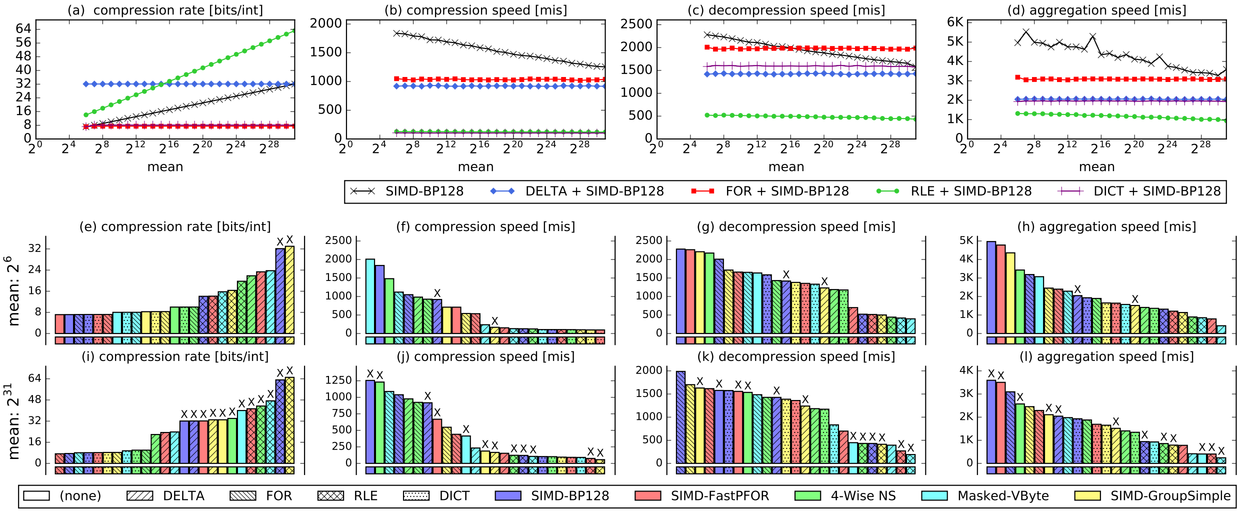

Figure 5: Comparison of the cascades on dataset D2.[5]

prove the data distribution in favor of NS. However, the data properties determine which techniques are suitable. In the worst case, the distributions might even become less suited. We also experimented with other data characteristics such as the number of distinct values and sorted datasets, but omit their results due to a lack of space. Those experiments lead to similar conclusions.

5.3 Cascades of Logical-Level and Physical-Level Techniques

To find out which improvements over the stand-alone NS algorithms the additional use of logical-level techniques can yield, we compare the five stand-alone NS algorithms from Section 5.1.2 to their cascades with the four logical-level techniques. That is, we compare $5 + 5 \times 4 = 25$ algorithms in total. The evaluation is conducted on three datasets: D1 and D5, which are already known, and D6, a *sorted* dataset for which we vary the number of distinct data elements by uniformly drawing values from the range $[0, max]$, whereby max starts with 0 and is increased until we reach 100 M distinct values, i.e., until all data elements are unique. For all three datasets, we provide a detailed comparison of SIMD-BP128 to its cascaded derivatives as well as a comparison of all 25 algorithms for selected data configurations. For our generic cascade algorithm, we chose a block size of 16 KiB, i.e., 4096 uncompressed integers. This size is a multiple of the block sizes of all considered algorithms and fits into the L1 cache of our machine. We also experimented with larger block sizes, but found that 16 KiB yields the best speeds.

Figure 5 (a-d) show the results of SIMD-BP128 and its cascaded variants on D2. The results for the compression rate are consistent with the distributions in Fig. 3 (h-l): Combined with FOR or DICT, SIMD-BP128 always yields equal or better results than without a preprocessing, while DELTA and RLE affect the results negatively. However, the cascades with logical-level techniques decrease the speeds of the algorithm, whereby the slow-down is significant for small data elements, but becomes acceptable for large values at least for DICT (decompression) and FOR. Indeed, the decompression of FOR + SIMD-BP128 is faster than SIMD-BP128 alone for means larger than 2^{16}. A comparison of all 25 algorithms can be found in Fig. 5 (e-h) and (i-l) for means of 2^6 respectively 2^{31}.[5] For the small mean, the cascades with RLE and DELTA achieve the worst compression rates, while for DICT, FOR and stand-alone NS, the algorithms are roughly grouped by the employed NS algorithm, since DICT and FOR do not change the distributions for the considered mean (see Fig. 3 (h-l)). Regarding the speeds, the top ranks are held by stand-alone NS algorithms. When changing the mean to 2^{31}, the cascades with FOR and DICT achieve by far the best compression rates. Stand-alone NS algorithms are still among the top ranks for the speeds. However, none of them achieves an actual size reduction. While depending on the application, many trade-offs between compression rate and speed could be reasonable, it generally does not make sense, to accept compression rates of more than 32 bits/int, since then, the data could rather be copied or not touched at all, which would be even faster. Keeping this in mind, the cascades with FOR achieve the best results regarding all three speeds, whereby DELTA also makes it into the top-3 for the compression.

Figure 6 shows the results on D5. The cascades of any logical-level technique and SIMD-BP128 achieve better compression rates than the stand-alone SIMD-BP128 from some run length on (Fig. 6 (a)). Regarding the (de)compression speeds, only RLE + SIMD-BP128 can yield an improvement, if the run length exceeds 2^5. It is noteworthy that the cascades with DELTA and FOR imply only a slight slow down, while they achieve much better compression rates. The aggregation speed of RLE + SIMD-BP128 gets out of scope for any other cascade for run lengths above 2^8, since the aggregation of RLE has to execute only one multiplication and one addition *per run*. The next three rows of Fig. 6 compare all cascades for average run lengths of 6, 37, and 517. Even for the lowest of these run lengths (Fig. 6 (e-h)),

[5] The bars in these diagrams are sorted, such that the best algorithm is at the left. We use the color to encode the NS algorithm and the hatch to encode the logical-level technique, whereby *(none)* means a stand-alone NS algorithm. Furthermore, bars with an X on top mark algorithms which do not achieve a size reduction on the dataset, i.e., require at least 32 bits per integer.

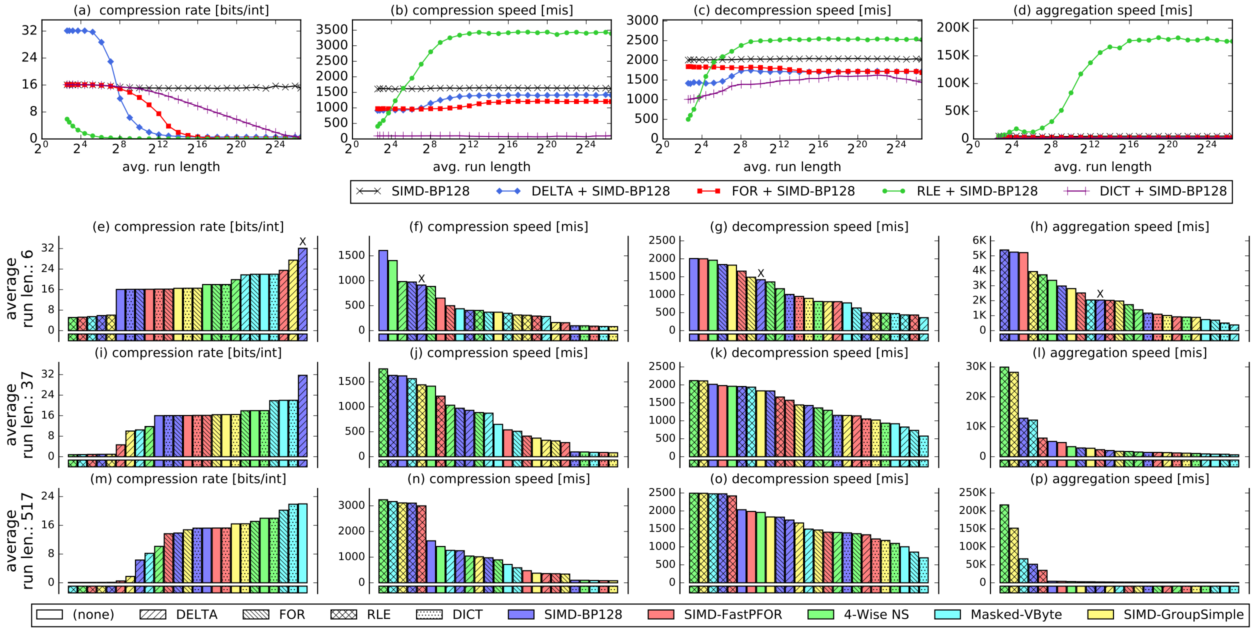

Figure 6: Comparison of the cascades on dataset D5.[5]

the cascades with RLE yield by far the best compression rates, while those with DELTA are among the last ranks. However, the (de)compression speeds of the cascades with RLE are not competitive to those of the best stand-alone NS algorithms. On the other hand, RLE + SIMD-BP128 has the best aggregation speed. As the run lengths get a little higher (Fig. 6 (i-l)), the cascades with RLE move further towards the top-ranks of the speeds and further improve their compression rates. Interestingly, the compression rates of the cascades with DELTA do now achieve the best compression rates after the cascades with RLE, except for DELTA + SIMD-BP128, which still yields the worst compression rate. When the run length is increased further (Fig. 6 (m-p)), these trends continue and the cascades with RLE do now dominate both, the compression rate and all three speeds.

Figure 7 (a-d) report the results of SIMD-BP128 and its cascades on D6 subject to the number of distinct data elements. Since D6 is sorted, a low number of distinct values is equal to a high average run length. Consequently, RLE + SIMD-BP128 achieves a better compression rate than stand-alone SIMD-BP128 until the number of distinct values comes close to the total number of values, i.e. 100 M. Although the possible minimum value is zero, FOR + SIMD-BP128 also improves the compression rate. This is due to the fact that within each input block of the cascade, the value range is small as the data is sorted. Apart from that, especially the decompression speed is interesting. For low numbers of distinct values and thus long runs, SIMD-BP128 and its cascade with RLE are nearly equally fast. As the number of distinct values increases, SIMD-BP128 is affected stronger than RLE + SIMD-BP128. However, when the number of distinct values exceeds 2^{21}, the performance of the cascade with RLE deteriorates and from this point on, the cascade with FOR, respectively DELTA is the fastest algorithm. Note that in this case, the decompression of the stand-alone SIMD-BP128 is never the fastest alternative. Figure 7 (e-h) show the comparison of all 25 algorithms when the dataset contains 128 distinct values. Since the average run length is very high (nearly 800k), the cascades including RLE are the best regarding both, compression rate and speeds. The extreme case of unique data elements, i.e., 100 M distinct values, is given in Fig. 7 (i-l). Now the cascades of RLE are among the worst algorithms for all four measured variables, since the data contains no runs. The best compression rates are now achieved by the cascades of DELTA, since the data is sorted. While the fastest compressor is stand-alone SIMD-BP128, the next ranks are held by cascades making use of DELTA. Regarding the decompression speed, the top-3 algorithms use SIMD-BP128 for the NS-part and DELTA, FOR, or no preprocessing. In terms of the aggregation speed, the stand-alone NS algorithms SIMD-BP128 and SIMD-FastPFOR are the fastest. However, DELTA + SIMD-BP128 and FOR + SIMD-BP128 also achieve very good aggregation speeds, but much better compression rates.

Summing up, the changes to the data distributions achieved by the logical-level techniques do indeed propagate to the compression rates of their cascades with NS. Furthermore, the speeds of the cascades can even exceed those of the corresponding stand-alone NS algorithms. This is especially true for the cascades including RLE, if the data contains long enough runs. Cascades with the other three logical-level techniques generally lead to less significant speed ups or even slow downs, whereby these often come with an improvement of the compression rate. Finally, if the logical-level technique is fixed, its cascades with different NS algorithms can lead to significantly different results regarding compression rate and speed. This justifies the consideration of multiple different NS algorithms even in cascades.

5.4 Lessons Learned

In order to employ lightweight compression effectively, it is desirable to know which algorithm is most suitable for a

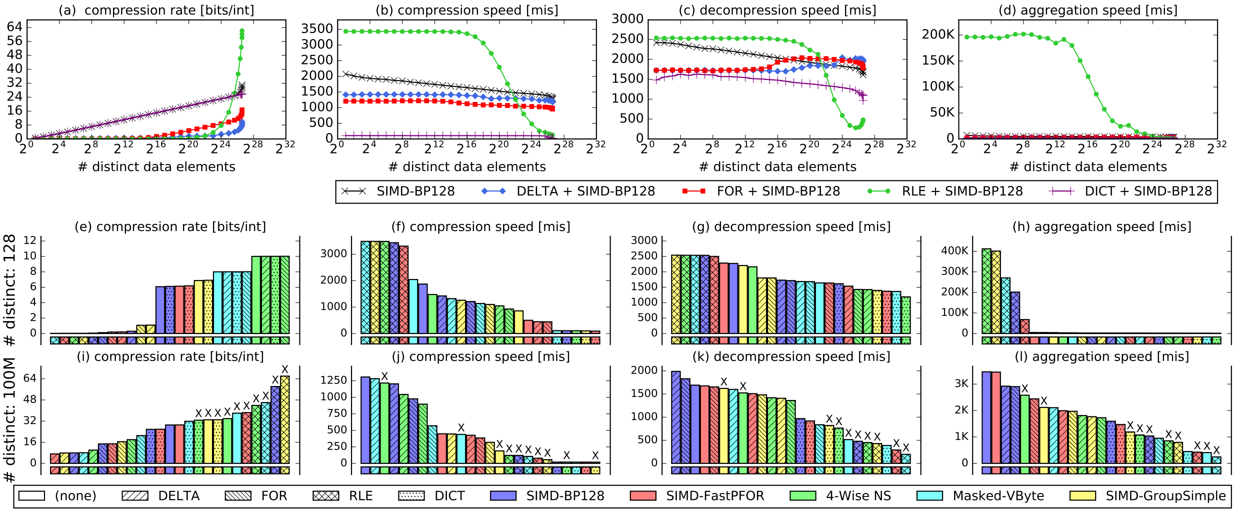

Figure 7: Comparison of the cascades on dataset D6.[5]

given data set w.r.t. a certain optimization goal as, e.g., the best compression rate, the highest (de)compression speed, or a combination thereof. Regarding the compression *techniques*, we can observe some general trends. For instance, NS usually performs the better, the lower the values are, while RLE profits from long runs, and DICT from few distinct values. However, these facts can be derived from the ideas of the techniques and have already been shown experimentally by other authors, e.g. in [1]. What is more interesting is the level of the compression *algorithms*. While SIMD-BP128 seems to be a good choice regardless of the optimization goal *if the data exhibits a good locality*, the case is more complicated for data with a low locality. What makes a decision even more complex is that the performance of some NS algorithms is not monotonic in the size of the values. This holds, e.g., for the word-aligned NS algorithms (Fig. 1 (i-l)) as well as Masked-VByte (Fig. 2, second column).

Moreover, lightweight data compression is still a hot research topic. Hence, more algorithms will be published in the future. Therefore, an automatic approach for choosing the best out of a set of algorithms would be welcome. It is self-evident that the naïve solution of first executing all considered algorithms on the exact data to be compressed and then choosing the best algorithm is infeasible for efficiency reasons. Instead, a model of the algorithms' compression rate and performance – subject to the data properties – could be used. While we believe that such a model could be built based on our systematic evaluation, we see the main contribution of this paper in illustrating that this decision is non-trivial and that, thus, further research in the direction of an automatic selection is necessary. However, since this is beyond the scope of this paper, we leave it for future work.

6. EXPERIMENTS ON REAL DATA

To complement our experiments on synthetic data, we evaluated the algorithms considered in Section 5.3 on a dataset of postings lists of the real-world document collection GOV2[6], which is frequently used to evaluate integer compression algorithms [12, 18, 20]. GOV2 is a corpus of 25 M documents found in a crawl of the `.gov` websites. The dataset contains about 1.1 M postings lists in total, each of which is a *sorted* array of *unique* 32-bit document ids. We discarded all lists containing less than 8192 integers, since time measurements are not reliable enough on too short arrays.

Figure 8 (a-d) compare SIMD-BP128 to its cascaded derivatives subject to the list length. Note that, as the total number of entries in the lists increases, so does the number of distinct entries (uniqueness), while the average difference of two subsequent entries decreases. The compression rate of RLE is noncompetitive to the other algorithms, since unique values imply the absence of runs. Employing any other logical-level technique can yield an improvement of the compression rate, whereby DELTA and FOR get better as the lists get longer. Regarding the compression and aggregation speed, pure SIMD-BP128 is the fastest for all list lengths. However, its cascades with DELTA respectively FOR (only aggregation) are not much slower for long lists. Regarding the decompression, using DELTA or FOR yields better results than pure NS for lists longer than 2^{19} respectively 2^{20}.

Figure 8 (e-h) provide the rankings of all 25 algorithms. The reported measurements are averages over all lists lengths weighted by the actual distribution of the lengths in the dataset. The cascades with DELTA yield the best compression rates. The fastest compressors are SIMD-BP128 and 4-Wise NS followed by their cascades with DELTA. Regarding the decompression and aggregation speeds, the top-4 are stand-alone NS algorithms, which are followed by cascades with DELTA or FOR.

7. CONCLUSION

Lightweight data compression is heavily employed by modern in-memory column-stores in order to compensate for the low main memory bandwidth. In recent years, the corpus of available compression algorithms has significantly grown, mainly due to the use of SIMD extensions. In our experimental survey, we systematically evaluated recent vectorized algorithms of all five basic techniques of lightweight compression as well as cascades thereof on a multitude of synthetic

[6] This data set of postings lists is provided by Lemire et al. at `http://lemire.me/data/integercompression2014.html`.

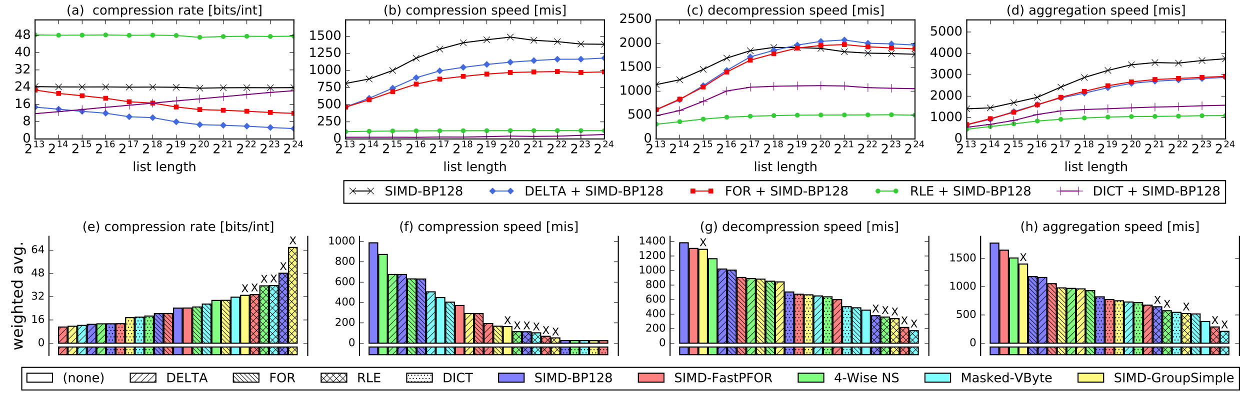

Figure 8: Comparison of the cascades on the postings lists of the real-world document collection GOV2.[5]

and one real data set. We have shown that there is no single-best algorithm suitable for all data sets. Instead, making the right choice is non-trivial and always depends on data properties such as value distributions, run lengths, sorting, and the number of distinct data elements. Furthermore, the best algorithm regarding the compression rate is often not the best regarding the (de)compression speed, such that a trade-off must be defined. Even the various null suppression algorithms show significantly different behavior depending on the data distribution. Finally, cascades of two techniques can heavily improve the compression rate, which comes at the cost of a lower speed in some, but not all cases.

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).

8. 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] D. Arroyuelo, S. González, M. Oyarzún, and V. Sepulveda. Document identifier reassignment and run-length-compressed inverted indexes for improved search performance. In *SIGIR*, 2013.

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

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

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

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

[8] A. Elgohary, M. Boehm, P. J. Haas, F. R. Reiss, and B. Reinwald. Compressed linear algebra for large-scale machine learning. *PVLDB*, 9(12), 2016.

[9] J. Goldstein, R. Ramakrishnan, and U. Shaft. Compressing relations and indexes. In *ICDE*, 1998.

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

[11] 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.

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

[13] D. Lemire, L. Boytsov, O. Kaser, M. Caron, L. Dionne, M. Lemay, E. Kruus, A. Bedini, M. Petri, and R. B. Araujo. The FastPFOR C++ library: Fast integer compression, `https://github.com/lemire/FastPFOR`.

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

[15] M. A. Roth and S. J. Van Horn. Database compression. *SIGMOD Rec.*, 22(3), 1993.

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

[17] F. Silvestri and R. Venturini. Vsencoding: Efficient coding and fast decoding of integer lists via dynamic programming. In *CIKM*, 2010.

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

[19] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Commun. ACM*, 30(6), 1987.

[20] 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.

[21] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theor.*, 23(3), 1977.

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