Patrick Damme, Dirk Habich, Wolfgang Lehner

A Benchmark Framework for Data Compression Techniques

Erstveröffentlichung in / First published in:

Performance Evaluation and Benchmarking: Traditional to Big Data to Internet of Things : 7th TPC Technology Conference. Kohala Coast, 31.08.-04.09.2015. Springer, S. 77–93. ISBN 978-3-319-31409-9.

DOI: http://dx.doi.org/10.1007/978-3-319-31409-9_6

Final edited form was published in "Performance Evaluation and Benchmarking: Traditional to Big Data to Internet of Things: 7th TPC Technology Conference. Kohala Coast 2015", S. 77–93, ISBN 978-3-319-31409-9
http://dx.doi.org/10.1007/978-3-319-31409-9_6

A Benchmark Framework for Data Compression Techniques

Patrick Damme(✉), Dirk Habich, and Wolfgang Lehner

Database Systems Group, Technische Universität Dresden,
01062 Dresden, Germany
{patrick.damme,dirk.habich,wolfgang.lehner}@tu-dresden.de

Abstract. Lightweight data compression is frequently applied in main memory database systems to improve query performance. The data processed by such systems is highly diverse. Moreover, there is a high number of existing lightweight compression techniques. Therefore, choosing the optimal technique for a given dataset is non-trivial. Existing approaches are based on simple rules, which do not suffice for such a complex decision. In contrast, our vision is a cost-based approach. However, this requires a detailed cost model, which can only be obtained from a systematic benchmarking of many compression algorithms on many different datasets. A naïve benchmark evaluates every algorithm under consideration separately. This yields many redundant steps and is thus inefficient. We propose an efficient and extensible benchmark framework for compression techniques. Given an ensemble of algorithms, it minimizes the overall run time of the evaluation. We experimentally show that our approach outperforms the naïve approach.

Keywords: Lightweight data compression · Main memory database systems · Efficient benchmarking

1 Introduction

Nowadays, main memory-centric column-oriented database management systems are the prevailing technology for data processing [5,7,12]. Many of these systems reduce the amount of data they have to store and process by making use of lossless lightweight compression [1,3]. Owing to its reduced size, 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 look-aside buffer. Moreover, many plan operators in database systems can be modified to directly process compressed data, which can be faster than processing the original data [1,16]. On the other side, compression and decompression introduce a certain computational overhead. For that reason, it is crucial to implement these algorithms as efficient as possible.

The data being stored and processed by in-memory column stores is as diverse as the application domains of such systems. It is characterized by a multitude of

Final edited form was published in "Performance Evaluation and Benchmarking: Traditional to Big Data to Internet of Things: 7th TPC Technology Conference. Kohala Coast 2015", S. 77–93, ISBN 978-3-319-31409-9
http://dx.doi.org/10.1007/978-3-319-31409-9_6

properties, such as data types, value distributions – including value ranges and the number of distinct values – and correlations between subsequent values – e.g., multiple subsequent occurrences of the same value. All these different kinds of data must be compressed appropriately, since databases are expected to be generic regarding the application area. At the same time, several decades of research in the field of lossless compression yielded a multitude of different compressed formats and compression algorithms to choose from. Some of these use completely different approaches, i.e., address different sources of redundancy in the original data, while others just differ in minor – but perhaps decisive – details of the memory layout of their outputs. Some are tailored to fit certain characteristics of modern hardware, while others are kept more generic.

When integrating compression into a database management system, it is crucial to know which compression technique is suited best for what data. In that respect, *suited best* can have different meanings ranging from *lowest latency* or *highest throughput* over *best compression rate* to *optimal for further processing by plan operator X*. Not leveraging such knowledge during the integration leads to a suboptimal improvement of the database performance, in the best case. It might, however, even cause a degradation of the performance, since most compression techniques increase the size of the data if it does not exhibit appropriate characteristics.

In the literature, several approaches exist attempting to identify a good compression technique for some given data using *rule-based* strategies. One example of these is the decision tree provided by Abadi et al. in [1]. However, rule-based approaches are too coarse-grained and can hardly capture all imaginable data characteristics. In contrast, we believe that a *cost-based* approach should be used instead. For this, an underlying cost model is required, which must be obtained from a *systematic* and *exhaustive* benchmarking and evaluation of numerous compression algorithms on multitudes of different original data and on different hardware. This systematic benchmarking is the scope of this paper. Thus, our contribution is a benchmark framework for data compression techniques with the following key features:

- It minimizes the overall evaluation run time for an ensemble of algorithms by identifying and eliminating redundant steps and making use of today's large main memories.
- It efficiently verifies the correctness of the results of the evaluated algorithms.
- It is highly extensible; in particular, it allows the integration of third-party compression algorithms and data generators, e.g., via a wrapper.

The rest of this paper is organized as follows: In the next section, we give an overview of existing lightweight compression techniques and state more precisely which classes of algorithms are relevant to our benchmarking. After that, Sect. 3 provides a high-level description of our benchmark framework. Section 4 discusses the execution of compression algorithms in detail. Our experimental results are presented in Sect. 5. We discuss related work in Sect. 6. Finally, we conclude our paper in Sect. 7.

Final edited form was published in "Performance Evaluation and Benchmarking: Traditional to Big Data to Internet of Things: 7th TPC Technology Conference. Kohala Coast 2015", S. 77–93, ISBN 978-3-319-31409-9
http://dx.doi.org/10.1007/978-3-319-31409-9_6

2 Compression Techniques

In the context of database-oriented data compression, there are three classes of algorithms relevant to our benchmark framework. The first two – compression and decompression – are discussed in Sect. 2.1. The third one – transformation – is discussed in Sect. 2.2. We only consider lossless techniques. Furthermore, we only investigate 32-bit integers as the data type of the uncompressed data at the current status. However, this limitation is not too strict, since integers can be obtained from values of any data type by applying dictionary coding [1,11], first. Our framework itself is generic in terms of data types.

2.1 Compression and Decompression

The general idea of compression is to represent some given original data in another format in which the data has a smaller size. Thereby, the information necessary to re-obtain the original data must be preserved, in order to allow a lossless decompression. There are two classes of compression techniques: heavyweight and lightweight. Heavyweight techniques, such as Huffman encoding [6], arithmetic coding [17], or LZW [10] compress the given data close to its entropy. However, they require a lot of computation and are, hence, rather suited for disk-centric DBMS. For main memory-centric systems, there are lightweight compression techniques which require much less computation. As a consequence, they achieve much higher throughputs, while still yielding good compression rates.

The basic types of lightweight compression are dictionary coding (DICT) [1,11,18], delta coding (DELTA) [9,13], frame-of-reference (FOR) [4,18], null suppression (NS) [1,13], and run-length encoding (RLE) [1,13]. DICT maps each distinct value to a unique key. DELTA and FOR represent each value as the difference to its predecessor or a certain reference value, respectively. The aim of these three techniques is to obtain a sequence of small integers from the original data. After that, a scheme from the family of NS is typically applied to achieve the actual compression. NS eliminates leading zeros in small integers. RLE compresses uninterrupted sequences of occurrences of the same value, so-called *runs*, by representing them by just one occurrence followed by the length.

Recent research in the field of lightweight compression has especially investigated the efficient implementation of these classic schemes on modern hardware. For instance, Zukowski et al. [18] introduced the paradigm of patched coding, which especially aims at the exploitation of pipelining in modern CPUs. Another promising direction is the vectorization of compression techniques by using SIMD instruction set extensions such as SSE and AVX. Numerous vectorized techniques have been proposed in recent years, e.g., in [9,14,15].

Besides compression and decompression algorithms for *different* compressed formats, we explicitly consider different *variants* of the compression to and decompression from *a single* compressed format. Such variants exist at both, the algorithmic and the implementation level. Two implementations of the same compression algorithm could, e.g., differ in whether they use SIMD extensions

Final edited form was published in "Performance Evaluation and Benchmarking: Traditional to Big Data to Internet of Things: 7th TPC Technology Conference. Kohala Coast 2015", S. 77–93, ISBN 978-3-319-31409-9
http://dx.doi.org/10.1007/978-3-319-31409-9_6

or not. We show an example of this case in Sect. 5. Furthermore, when composing two distinct compression algorithms, e.g., DELTA followed by NS, there are several variants regarding the degree of integration. Our benchmark framework especially optimizes the comparison of different variants of one algorithm.

2.2 Transformation

Beyond compression and decompression, we are also concerned with a third class of algorithms: *transformation*, which we recently introduced in [2]. A transformation is a lossless change of the (compressed) format some data is represented in. It expects data of a certain *source format* and outputs the representation of that data in its *destination format*. Thus, transformation is a generalization of compression and decompression. Henceforth, we use the term transformation to refer to those transformations which are neither a compression nor a decompression, i.e., that do not involve uncompressed data as their input or output.

There are two possible ways to transform some data represented in a compressed format A to a compressed format B. A naïve approach, which we call *indirect transformation*, first applies the decompression algorithm of its source format to its entire compressed input data and materializes the result in main memory. After that, it uses the compression algorithm of its destination format. While indirect transformations can easily be implemented for arbitrary pairs of a source and a destination format, they are very inefficient due to their many accesses to main memory. In contrast, there are *direct transformation* techniques which do not materialize any intermediate data in main memory. Instead, intermediate data stays in CPU registers or at worst in the L1 cache. In general, direct transformations can be implemented by interleaving the code of the decompression of the source format with the compression of the destination format. Thereby, the intermediate store and load instructions are omitted. In many cases, even more optimizations are possible, which we described in detail in [2] for some direct transformation techniques. In [2] we also experimentally showed that direct transformations outperform their indirect counterparts in most cases.

Our benchmark framework supports the evaluation of transformation algorithms and makes use of them, in order to minimize the overall evaluation run time. How this is done is described in detail in Sect. 4.

3 Framework Overview

In this section, we provide a high-level view of our benchmark framework[1]. Section 3.1 describes the general workflow of the framework. After that, Sect. 3.2 explicitly explains our design principles.

[1] The source code of our framework can be downloaded at https://wwwdb.inf.tu-dresden.de/team/staff/patrick-damme-msc/.

Final edited form was published in "Performance Evaluation and Benchmarking: Traditional to Big Data to Internet of Things: 7th TPC Technology Conference. Kohala Coast 2015", S. 77–93, ISBN 978-3-319-31409-9
http://dx.doi.org/10.1007/978-3-319-31409-9_6

3.1 General Workflow

The general workflow of our benchmark framework is shown in Fig. 1. The applications performing the individual steps are invoked by a script. In the following, we briefly present each step.

Benchmark Specification. Before the framework can start, the user needs to specify a benchmark. She decides which algorithms shall be evaluated and subdivides these into several, possibly overlapping sets. The algorithms in one set will be evaluated on the same data. For each set of algorithms, the user specifies the data characteristics by choosing (1) a data generator, (2) one data property to be varied, including the distinct values to be assigned to it, and (3) the configuration of the remaining, fixed data properties. For instance, the user could specify to generate $100M$ values using generator `simpleGen`, whereby the values follow a uniform distribution over the interval $[2^8,\ 2^{16}-1]$, while the data consists of runs, whose lengths follow a normal distribution with a standard deviation of 5 and a mean that is varied from 20 to 200 in steps of 10. Varying a data property allows to investigate the influence of that property on the performance of the algorithms. If the user wants to vary more than one data property or she wants one data property to be varied in conjunction with multiple configurations of fixed properties, then she must provide multiple specifications of the input data for that set of algorithms, since only one property can be varied at a time. The specification of the benchmark is passed to the framework as a file with a very simple text-based format, an example of which is presented in Fig. 2.

Parsing. The first step of the framework is parsing the benchmark specification. After that, it knows which algorithms the user wants to evaluate on what data.

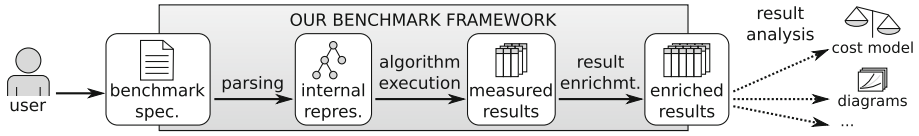

Fig. 1. An overview of our benchmark framework.

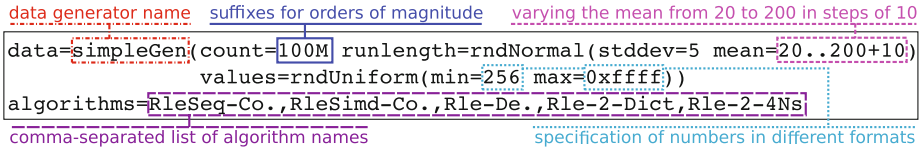

Fig. 2. Example benchmark specification using our minimalistic specification language. One benchmark may contain several of such pairs of data and algorithm specifications.

Final edited form was published in "Performance Evaluation and Benchmarking: Traditional to Big Data to Internet of Things: 7th TPC Technology Conference. Kohala Coast 2015", S. 77–93, ISBN 978-3-319-31409-9
http://dx.doi.org/10.1007/978-3-319-31409-9_6

Algorithm Execution. This is the core step of the framework. It is invoked once for each specified pair of a set of algorithms and a configuration of data properties. It iterates over all values to be assigned to the varied data property. In each iteration, the data is generated using the specified generator parameterized with the specified data properties. After that, the specified algorithms are run on the generated data. The framework also checks the output buffers of the algorithms for correctness. The approach the evaluation follows is crucial for its efficiency. We describe a naïve as well as our sophisticated approach in detail in Sects. 4.1 and 4.2, respectively. Since performance is crucial in this step, it is implemented in C++. The output is a set of CSV files. Each of these contains some meta data regarding the evaluation and a table containing only actual measurements, i.e., no derived results. For instance, for all algorithms we measure their *run time* and for each format, we measure the *size* of the data in that format.

Result Enrichment. In this step, the measured results output by the previous step are enriched with derived columns. We differentiate between two types of enrichment. The first one provides additional representations of the measurements for a single algorithm or compressed format. Examples include the *execution speed* of all algorithms or *compression rates* for all formats. The second kind of enrichment is concerned with the relation between the measurements of different algorithms or formats. These could, for instance, be *speed ups* of one algorithm compared to another one. Separating the result enrichment from the algorithm execution was an explicit decision. That way we can easily calculate new derived results from *existing measurements without having to re-run the evaluation* at a later point in time *as a part of the framework's workflow.*

Result Analysis. After the benchmark framework is done, the collected results can be analyzed. Our vision is the creation of a cost model for the investigated compression, decompression, and transformation algorithms. The analysis could, however, also simply be the generation of diagrams visualizing the results.

3.2 Design Principles

When designing our benchmark framework, we tried to adhere to three major design principles:

Simplicity. Our framework uses a simple and yet expressive language to specify a benchmark. Instead of describing it in detail, we just provide an example in Fig. 2. Moreover, the steps of the framework are easily repeatable. Once a benchmark has been specified, it can be run arbitrarily often, e.g., on different hardware.

Efficiency. The framework minimizes the overall run time of the evaluation. Our main focus is on the avoidance of redundant steps during the algorithm execution. To achieve this, we utilize today's large main memories by keeping outputs instead of recomputing them. Details are provided in Sect. 4.

Final edited form was published in "Performance Evaluation and Benchmarking: Traditional to Big Data to Internet of Things: 7th TPC Technology Conference. Kohala Coast 2015", S. 77–93, ISBN 978-3-319-31409-9
http://dx.doi.org/10.1007/978-3-319-31409-9_6

Extensibility. The framework can be extended at several crucial points. In particular, it allows the integration of third-party compression, decompression and transformation algorithms. The framework defines a common interface for these. To use an existing algorithm with our framework, the algorithm must implement that interface. Alternatively, a wrapper can be provided which translates our interface to that of the third-party algorithm. In fact, we implemented a wrapper for the compression and decompression algorithms of the FastPFor library by Lemire et al. [8]. In a similar way, third-party data generators can be integrated. One particular extension could be a data generator which does not produce synthetic data, but provides the system with data from a real world dataset.

4 Efficient Execution of Compression Techniques

Since it is the most critical step for the evaluation efficiency, the *algorithm execution* from Sect. 3.1 is explained in detail here. Its input is a specification of the data properties along with a set $\mathcal{A}$ of algorithms to be evaluated on that data. We assume that (1) the original data has a non-trivial size, e.g., several hundreds of megabytes, and (2) the value of one data property is varied within a range of a non-negligible cardinality, e.g., several hundreds or thousands of different values. Otherwise, the evaluation would not take much time and therefore not offer much potential for optimization.

When applying a compression or transformation algorithm to some data, it must be ensured that a sufficiently large output buffer was allocated to prevent buffer overflows. However, in general, it is impossible to know the *exact* size of the output beforehand. Thus, a *pessimistic estimation* of the result size is required. If nothing about the characteristics of the original data is known, such an estimation can only be based on the number of original values. Unfortunately, for most compressed formats, the size of the data increases, in the worst case. For instance, data compressed with RLE is twice as large as the uncompressed data, if it does not contain any run of a length greater than one. As a consequence, the system could end up allocating to much memory. While this might be a problem in a DBMS, we can circumvent it in our framework, since we know the properties of the original data from the user's specifications. Our estimation uses this information to find a tighter fit for the actual result size, while never underestimating it. Note, that this information is not made available to the algorithms under investigation, since this could be considered unfair.

During the evaluation, we also check the result of each algorithm for correctness. Each result is compared to the original data, which might require a decompression, first. Moreover, we place some random bytes after the end of each output buffer, in order to be able to detect buffer overflows. Note that we assume that all algorithms have been designed, implemented, and tested carefully *before* the benchmark. Hence, the purpose of the checks is only the detection of errors, not the generation of detailed information for debugging.

While a certain overhead for the evaluation cannot be avoided, different evaluation approaches differ in the implied amount of overhead. In the following, we

Final edited form was published in "Performance Evaluation and Benchmarking: Traditional to Big Data to Internet of Things: 7th TPC Technology Conference. Kohala Coast 2015", S. 77–93, ISBN 978-3-319-31409-9
http://dx.doi.org/10.1007/978-3-319-31409-9_6

Table 1. The notations used in this section.

Symbol	Meaning
S, D, X, Y; U	arbitrary formats; the uncompressed format
O; O_X	the original data; the representation of O in the format X
A; $\mathcal{A}$	an algorithm; the set of all algorithms specified by the user

describe a naïve evaluation approach in Sect. 4.1. After that, we present our sophisticated approach in Sect. 4.2. Finally, we compare the two of them in Sect. 4.3. Table 1 presents the notations used in those sections. As a running example, we assume the user wants to benchmark two variants of the compression of RLE (*RleSeq-Co.*, *RleSimd-Co.*), the decompression of a dictionary scheme (*Dict-De.*) as well as the transformation from the format of 4-Wise Null Suppression [14] to that of 4-Gamma Coding [14] and vice versa (*4Ns-2-4G*, *4G-2-4Ns*).

4.1 A Naïve Evaluation Approach

When conducting the benchmark naïvely, every algorithm is evaluated in isolation, i.e., as if it was the only algorithm to be investigated. For each algorithm A in $\mathcal{A}$ with source format S and destination format D, the following steps have to be performed for *every* value to be assigned to the varied data property:

1. Generate the original data O_U using the specified data generator and data characteristics, including the current value of the varied data characteristic.
2. If $S \neq U$: Obtain O_S by applying a compression algorithm for S to O_U.
3. Apply A to O_S . Let the output of A be R_D.
4. If $D \neq U$: Obtain R_U by applying a decompression algorithm for D to R_D.
5. Compare R_U and O_U byte by byte to check whether all involved algorithms worked correctly.

Figure 3 illustrates these steps for the example $\mathcal{A}$ given above. This procedure is very simple and thus easy to implement. However, it is very inefficient in general, since it implies multiple invocations of the same algorithms on the same

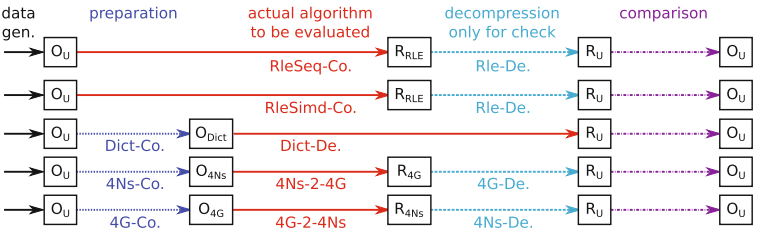

Fig. 3. The steps of a naïve evaluation strategy.

Final edited form was published in "Performance Evaluation and Benchmarking: Traditional to Big Data to Internet of Things: 7th TPC Technology Conference. Kohala Coast 2015", S. 77–93, ISBN 978-3-319-31409-9
http://dx.doi.org/10.1007/978-3-319-31409-9_6

data. In our example, the decompression of RLE is applied twice and the data generation is done five times using the same properties.

4.2 Our Sophisticated Evaluation Approach

The basic idea of our sophisticated approach is the avoidance of redundant steps in the evaluation. That is, no algorithm is invoked twice with the same input. This holds for the algorithms and for the data generation. The key to this is keeping the result buffers of all algorithms, *as long as they are needed* by another algorithm being executed later. These are the high-level steps of our approach:

1. Find out which additional algorithms are needed for preparations and checks.
2. Determine a valid execution order of the algorithms.
3. Run the algorithms one by one and take the measurements.

Step 3 is re-executed for each value to be assigned to the varied data property, step 2 is only re-run if otherwise the memory consumption would be too high and step 1 is done only once in advance for $\mathcal{A}$. In the following, we explain each of these steps in detail. Figure 4 illustrates them for the example given above.

Finding Required Additions for Preparations and Checks. For this step, we formalize the benchmark as a directed graph. The vertices of that graph represent formats, while an edge (X, Y) represents an algorithm with source format X and destination format Y. Since we explicitly consider different variants of a certain algorithm, multiple edges between two nodes are allowed.

First, we build a graph representing the user's benchmark. For each algorithm A in $\mathcal{A}$ with source format S and destination format D, we add the two nodes S and D as well as an edge (S, D) to the graph. That is, we initialize the graph such that it contains only the algorithms in $\mathcal{A}$ as well as the formats involved in them. If the uncompressed format is not already included as a node, we explicitly add it. The result of this step for our example $\mathcal{A}$ is shown in Fig. 4 (a).

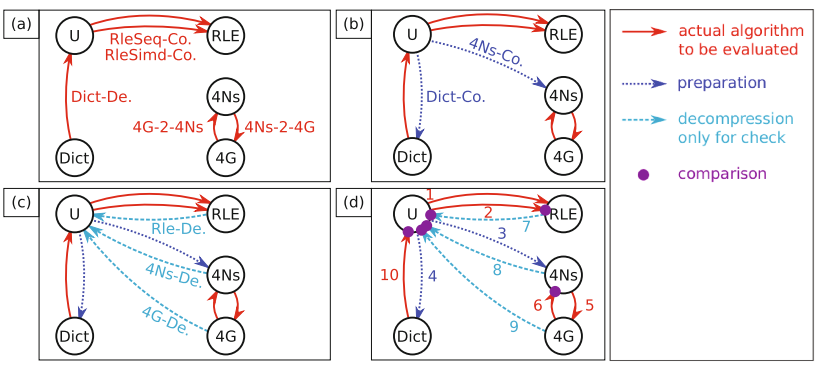

Fig. 4. The steps of our sophisticated evaluation strategy.

Final edited form was published in "Performance Evaluation and Benchmarking: Traditional to Big Data to Internet of Things: 7th TPC Technology Conference. Kohala Coast 2015", S. 77–93, ISBN 978-3-319-31409-9
http://dx.doi.org/10.1007/978-3-319-31409-9_6

Next, we ensure that the source format of each algorithm A in $\mathcal{A}$ can actually be reached from the uncompressed format by a sequence of transformations. This is important, since otherwise the input data for A might not be available. To achieve this, we iterate the following two steps until all nodes are reachable (see Fig. 4 (b) for a possible resulting graph for our example $\mathcal{A}$):

1. *Identify an unreachable node X.* If the graph contains a source, i.e., a node without ingoing edges, we select this one. If there are no sources, there could still be nodes on cycles unreachable from the node of the uncompressed format. To find these, we perform a cycle-aware depth-first search. If the search does not find all nodes, we randomly pick one of the nodes not found.
2. *Make X reachable.* For doing so, there could be multiple possibilities. For simplicity, we always connect X by adding a compression from U to X.[2]

Since we also want to check the algorithms in $\mathcal{A}$ for correctness, we need to decompress the representation of the data in each involved format in order to be able to compare it to the original data. Consequently, for each node in the graph we add an edge to the node of the uncompressed format, *if there is not already one*. The result of this step for our example $\mathcal{A}$ is depicted in Fig. 4 (c). We denote the set of algorithms resulting from this step by $\mathcal{A}^+$.

Determining the Execution Order. After it is clear which additional algorithms must be performed, the second step is to determine the order in which the algorithms shall be applied. A valid execution order must fulfil two requirements: (1) it must contain each algorithm in $\mathcal{A}^+$ exactly once, and (2) before an algorithm with a *source* format X other than the uncompressed format is applied, an algorithm with *destination* format X must have been applied. The latter condition ensures that the appropriate input data is available for each algorithm, whereby in the beginning, only the uncompressed data is present.

One strategy to obtain a valid order is the following: First, all compressions in $\mathcal{A}^+$ are executed. Second, all transformations are run. Finally, all decompressions are applied.[3] One possible outcome for our example $\mathcal{A}$ is shown in Fig. 4 (d). For general $\mathcal{A}^+$, this strategy requires a lot of main memory, since the compressed buffers of each format allocated in the first phase must be kept in memory until after the respective decompression in the third phase. Thus it is obvious that – while it does not affect the run time – the execution order is decisive for the

[2] Note that, alternatively, a transformation from some already reachable node to X could be added. This could be especially useful, since transformations are faster than compressions in many cases. However, finding the fastest way to make X reachable would require a cost model for the algorithms, which can only be available *after* the systematic benchmarking.

[3] The compressions can be executed in an arbitrary order. The same applies to the decompressions. However, the transformations cannot be applied in an arbitrary order in general, since a transformation could require a source format that is not present after all compressions in $\mathcal{A}^+$ have been executed, as it is the case for 4G-2-4Ns in our example.

Final edited form was published in "Performance Evaluation and Benchmarking: Traditional to Big Data to Internet of Things: 7th TPC Technology Conference. Kohala Coast 2015", S. 77–93, ISBN 978-3-319-31409-9
http://dx.doi.org/10.1007/978-3-319-31409-9_6

maximum memory requirement of the evaluation. There are simple memory-efficient solutions for certain special cases. For instance, if $\mathcal{A}^+$ contains only compressions and decompressions, we simply determine the order the following way: We process the compressed formats involved in $\mathcal{A}^+$ one by one. For each compressed format X, we first execute all compressions to X in $\mathcal{A}^+$, then we perform all decompressions from X in $\mathcal{A}^+$. That way, there is no point in time when O_X and O_Y are both in memory, for any compressed formats X and Y, with $X \neq Y$. If $\mathcal{A}^+$ also contains transformations, finding an optimal order – or a sufficiently good order with respect to some upper bound for the maximum memory consumption – is non-trivial and might itself take considerable time. However, our framework can easily be extended at that point by implementing an appropriate order strategy.

Evaluating the Algorithms. When the order of the algorithms is fixed, the actual evaluation begins. In this step, two auxiliary data structures are maintained: (1) a map m mapping each format X to a buffer containing O_X, and (2) a set of *reference counters*, one for each format involved in $\mathcal{A}^+$. Initially, m is empty and for each format X involved in $\mathcal{A}^+$, the reference counter of X is the number of algorithms in $\mathcal{A}^+$ with source format X plus the number of those with destination format X. Thus, the reference counter of X is the number of times, O_X is needed either as input data or for a comparison. In the beginning, a buffer for the original data is allocated. Then the original data is generated according to the user's specifications and put into m with the key U.

Then, the algorithms are executed. A loop iterates over all A in $\mathcal{A}^+$ with source format S and destination format D using the order determined in the previous step and does the following:

1. A buffer b that fits O_D is allocated.
2. A is executed on O_S, which is looked up in m. Note that O_S is always available if the execution order is valid.
3. It is checked, whether m already contains a buffer for D.
   (a) If this is not the case, i.e., if A is the first algorithm producing a copy of O_D, then b is put in m with key D.
   (b) Otherwise, i.e., if there is already a copy of O_D, the content of b is compared to that copy byte by byte. If they are not the same, this means an error at some point. After the comparison, b is released.
4. The reference counters of both, S and D are decremented. If any of them reaches 0, i.e., if no algorithm executing later will need them any more, then the corresponding buffer is fetched from m and released to free memory.

This procedure guarantees that each required buffer is allocated as late as possible and released as early as possible. Note that all checks are covered by step 3b. If D is a compressed format, the check is performed *in the compressed space*. If D is the uncompressed format, it is not possible that step 3a is executed. This is due to the fact that O_U is in m right from the beginning onwards. It might be released before the end of the evaluation, but not before the last decompression,

Final edited form was published in "Performance Evaluation and Benchmarking: Traditional to Big Data to Internet of Things: 7th TPC Technology Conference. Kohala Coast 2015", S. 77–93, ISBN 978-3-319-31409-9
http://dx.doi.org/10.1007/978-3-319-31409-9_6

otherwise the reference counter for U cannot become 0. Thus, a decompression is always followed by a check with the uncompressed data.

4.3 Comparison of the Two Approaches

Our sophisticated approach avoids redundant executions of algorithms at several points. In particular, these are:

- The naïve approach generates the original data anew *for each algorithm* in $\mathcal{A}$. Our approach generates it *only once* for the entire ensemble of algorithms.
- If multiple algorithms in $\mathcal{A}$ share the same compressed source format X, then the naïve approach performs a compression to X *for each of these*, while our approach performs it *just once*. If some compression to X is already included in $\mathcal{A}$, our approach causes *no overhead* for the preparation regarding X.
- If several algorithms in $\mathcal{A}$ share the same compressed destination format Y, then the naïve approach executes a decompression from Y *for each of these*, whereas our approach runs it *just once*. Again, if this decompression is already part of $\mathcal{A}$, our approach causes *no overhead*.
- The naïve approach performs all checks in the uncompressed space, i.e., compares large buffers. In contrast, our approach does as many checks as possible in the compressed space, i.e., compares smaller compressed buffers.

To summarize these points, our approach is especially well-suited if $\mathcal{A}$ contains many algorithms with shared source and/or destination formats. This is, for instance, the case when different variants of the same algorithm are evaluated. Even if none of the algorithms in $\mathcal{A}$ have any formats in common, our approach can still save the time for the redundant data generation. If $\mathcal{A}$ contains only one algorithm, then both approaches do exactly the same. We investigate the quantitative differences between the two approaches in the next section.

5 Experimental Evaluation

We implemented our framework as well as several compression, decompression, and transformation algorithms in C++ and compiled them with `g++` 4.8 using the optimization flag `-O3`. Our experiments are conducted on a machine equipped with an Intel Core i7-4710MQ CPU at 2.5 GHz and 16 GB of RAM. Section 5.1 focuses on the comparison of the naïve and our sophisticated evaluation approach. After that, Sect. 5.2 reports some results output by our framework.

5.1 Naïve Approach vs. Our Sophisticated Approach

We compare the naïve approach as described in Sect. 4.1 (N) to our sophisticated approach as described in Sect. 4.2 (S). Regarding the execution order in S, we use the second idea described in Sect. 4.2 whenever possible and the first idea, otherwise. Additionally, we investigate a variant of the naïve approach that generates the original data only once for the ensemble of algorithms (N^+),

Final edited form was published in "Performance Evaluation and Benchmarking: Traditional to Big Data to Internet of Things: 7th TPC Technology Conference. Kohala Coast 2015", S. 77–93, ISBN 978-3-319-31409-9
http://dx.doi.org/10.1007/978-3-319-31409-9_6

Table 2. The example benchmarks used in this section. **C**, **D**, and **T** stand for compression, decompression, and transformation algorithms, respectively.

Benchmark	Varied property	#variations	Algorithms
B1	avg. run length	1.000	**C**: RleSeq, RleSimd
B2	max. bit width	24	**C**: 4Ns, 4G, Dict; **D**: 4Ns, 4G, Dict
B3	avg. run length	200	**C**: RleSeq, RleSimd; **D**: Dict; **T**: Rle-2-Dict, 4Ns-2-4G, 4G-2-4Ns
B4	max. bit width	24	**C**: 4G; **D**: Dict; **T**: 4Ns-2-Rle
B5	avg. run length	1.000	**C**: RleSimd

since this is an obvious and easy-to-implement improvement of N. We define five different example benchmarks (B1 to B5), representing different cases of sets of algorithms. Their specifications are summarized in Table 2. We consider the compression schemes 4-Wise NS (4Ns) [14], 4-Gamma (4G) [14], dictionary coding (Dict), a sequential and our vectorized variant of RLE (RleSeq and RleSimd) as well as transformations between these. The original data consists of $125M$ uncompressed 32-bit integers, i.e., 500 MB for each benchmark. We ran each of the three evaluation approaches for each of the five example benchmarks.

Figure 5 (a) presents the overall run times of the benchmarks as well as the time spent on the *specified* algorithms, the data generation, and the remaining overhead. Naturally, the execution of the algorithms specified in the benchmark takes equally much time, independent of the evaluation approach. N^+ and S need the same time for the data generation, since they generate it only once for the entire ensemble per value of the varied data property. In contrast, N re-generates the data for each algorithm, which is its major inefficiency. Due to that, it is never faster than any of the other two. The remaining overhead, i.e., preparations, decompressions added by the framework, and checks, is subject to the approach and the ensemble of algorithms. When $\mathcal{A}$ contains many algorithms with formats they have in common (B1 to B3), S can eliminate the redundant steps and thus outperforms the other two approaches. These cases are especially relevant to us. The more algorithms with formats they have in common $\mathcal{A}$ contains, respectively the longer these take, the higher the speed up of S. In B4, the specified algorithms have no compressed formats in common. Additionally, a compressed format is only used as the input of a transformation in $\mathcal{A}$, namely 4Ns. In this special case, S is slightly slower than N^+, because S also performs a check for 4Ns, which N^+ does not. In all other cases, S is not slower than any of the other two approaches. If only one algorithm is evaluated (B5), all three approaches require the same amount of time, since they do exactly the same then.

Figure 5 (b) reports the maximum memory consumption of each of the approaches during each of the benchmarks. It can be seen that the speed up of S in B1 and B3 is bought at the cost of a higher memory demand. However, in B2, S outperforms N and N^+ without requiring more memory. This is due to the execution order used by S in B2. Since $\mathcal{A}$ contains only one compression

Final edited form was published in "Performance Evaluation and Benchmarking: Traditional to Big Data to Internet of Things: 7th TPC Technology Conference. Kohala Coast 2015", S. 77–93, ISBN 978-3-319-31409-9
http://dx.doi.org/10.1007/978-3-319-31409-9_6

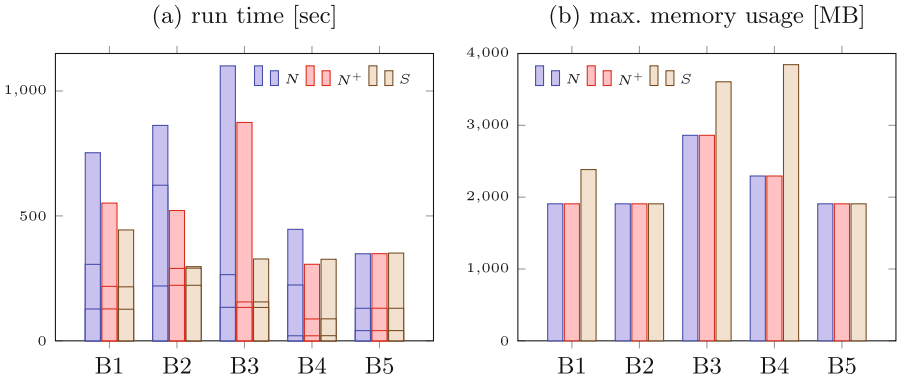

Fig. 5. A comparison of the three evaluation approaches for the different example benchmarks: (a) run times, subdivided into the time consumed by the specified algorithms (bottom), the data generation, and the remaining overhead (top), (b) maximum memory consumptions.

and one decompression algorithm for some formats, S does not need to keep the compressed buffers of any format longer than N and N^+. However, in certain cases (B4) S requires more memory without being faster.

5.2 Example Benchmark

Independent from the evaluation approach, our framework gives us insights into the performance of the investigated algorithms. To show just one example of the outputs of our framework, we choose B3 from the previous section. In B3, the data was generated such that it consists of runs. The individual run lengths were chosen uniformly from the interval $[x - 5, x + 5]$ whereby x was varied from 6 to 404 in steps of 2. The run values are uniformly distributed within the interval $[0, 255]$. In order not to overload the diagrams, we limit the presentation to our two variants of the compression of RLE (RleSeq and RleSimd) as well as the transformations 4Ns-2-4G and 4G-2-4Ns, respectively their involved formats. Figure 6 (a) reports the speeds of the algorithms. It can be seen that the speeds of RleSeq and RleSimd are subject to the run length, while those of the transformations between 4Ns and 4G are not. Furthermore, the vectorized algorithm RleSimd outperforms the sequential one for all run lengths. Figure 6 (b) presents the achieved compression rates of the formats. As expected, the compression rate of RLE decreases, i.e., gets better, as the run length increases. Interestingly, the compression rate of the format of 4-Gamma [14] is also slightly better for longer runs. This is due to the fact that 4G uses a shared bit width for four values at a time. With longer runs, there are more blocks consisting of four equal values, which have the same bit width, and thus waste less bits.

Final edited form was published in "Performance Evaluation and Benchmarking: Traditional to Big Data to Internet of Things: 7th TPC Technology Conference. Kohala Coast 2015", S. 77–93, ISBN 978-3-319-31409-9
http://dx.doi.org/10.1007/978-3-319-31409-9_6

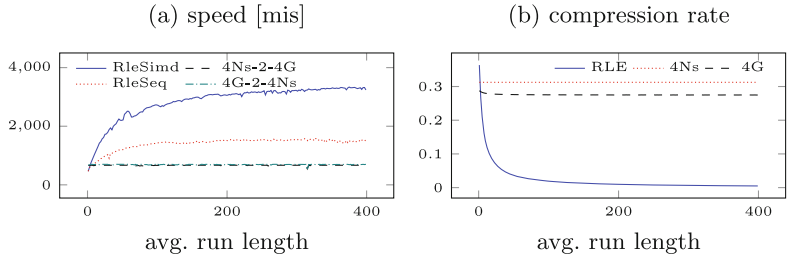

Fig. 6. The processing speeds in million integers per second (mis) (a) and compression rates (b) of some of the algorithms and formats included in B3.

6 Related Work

While we intend to base the decision for a compression technique on a cost model created from the output of our framework, there already exist other approaches to identify an optimal technique in the literature.

Abadi et al. [1] integrate several compression techniques into the column-oriented DBMS C-Store. Their aim is to improve the query performance by compressing every column appropriately. From their experiments, they *manually* derive a decision tree, which is based on certain data characteristics and on the access patterns of a column. However, they consider only a small number of compression schemes. We intend to consider a high number of techniques and *automatically* create a cost model.

Another interesting approach was proposed by Paradies et al. [11]. They intend to decrease the space requirement of the column-oriented SAP BW Accelerator. Being committed to dictionary coding, their goal is not to decide *how*, but *what* to compress. They observe that correlated columns can be stored more efficiently if their corresponding values are coded in pairs. To identify the most promising pairs of correlated columns, they propose an efficient algorithm, which employs sampling and pruning and is aware of additional compression techniques already applied to the data.

7 Conclusions and Future Work

We described our highly extensible benchmark framework for compression, decompression, and transformation algorithms. Based on a benchmark specification by a user, it generates synthetic data with certain characteristics and automatically runs the specified algorithms on it. We proposed a highly efficient evaluation approach, which is based on the elimination of redundant steps during the benchmark execution and thus minimizes the overall benchmark run time. Our experiments proved the superiority of our approach over a naïve approach.

Our ultimate research goal is to employ lightweight compression for the intermediate results in an in-memory DBMS and to make compression subject to query optimization. We intend to implement compression, decompression, and

Final edited form was published in "Performance Evaluation and Benchmarking: Traditional to Big Data to Internet of Things: 7th TPC Technology Conference. Kohala Coast 2015", S. 77–93, ISBN 978-3-319-31409-9
http://dx.doi.org/10.1007/978-3-319-31409-9_6

transformation as plan operators. Like for any other operator, the query optimizer must be able to estimate their costs (additionally, it needs to know, how efficient other plan operators can process the compressed data of a certain format). Simply knowing the best compression scheme for a given dataset does not suffice in our case, since we will need to know the overall costs of multiple alternative query executions plans, each of which might contain compression operators at several points. Hence, we need a cost model for compression techniques. To obtain this cost model, we need to benchmark a high number of algorithms on many different datasets in a structured and highly efficient way. Our framework enables us to do precisely that. Besides the execution of a single compression algorithm, we also intend to consider compositions of such algorithms as well as direct transformations between two formats. Both of these further increase the size of the benchmark space and thus stress the importance of our framework.

Acknowledgments. This work was 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).

References

1. Abadi, D., Madden, S., Ferreira, M.: Integrating compression and execution in column-oriented database systems. In: SIGMOD, pp. 671–682 (2006)
2. Damme, P., Habich, D., Lehner, W.: Direct transformation techniques for compressed data: general approach and application scenarios. In: Morzy, T., Valduriez, P., Ladjel, B. (eds.) ADBIS 2015. LNCS, vol. 9282, pp. 151–165. Springer, Heidelberg (2015)
3. Färber, F., May, N., Lehner, W., Große, P., Müller, I., Rauhe, H., Dees, J.: The SAP HANA database - an architecture overview. IEEE Data Eng. Bull. **35**(1), 28–33 (2012)
4. Goldstein, J., Ramakrishnan, R., Shaft, U.: Compressing relations and indexes. In: ICDE, pp. 370–379 (1998)
5. Große, P., Lehner, W., May, N.: Advanced analytics with the SAP HANA database. In: DATA, pp. 61–71 (2013)
6. Huffman, D.: A method for the construction of minimum-redundancy codes. Proc. IRE **40**(9), 1098–1101 (1952)
7. Lee, J., Kwon, Y.S., Färber, F., Muehle, M., Lee, C., Bensberg, C., Lee, J., Lee, A.H., Lehner, W.: SAP HANA distributed in-memory database system: transaction, session, and metadata management. In: ICDE, pp. 1165–1173 (2013)
8. Lemire, D., Boytsov, L., Kaser, O., Caron, M., Dionne, L., Lemay, M., Kruus, E., Bedini, A., Petri, M.: The FastPFOR c++ library: Fast integer compression. https://github.com/lemire/FastPFOR
9. Lemire, D., Boytsov, L.: Decoding billions of integers per second through vectorization (2012). CoRR abs/1209.2137
10. Lempel, A., Ziv, J.: On the complexity of finite sequences. IEEE Trans. Inf. Theory **22**, 75–81 (1976)
11. Paradies, M., Lemke, C., Plattner, H., Lehner, W., Sattler, K.U., Zeier, A., Krueger, J.: How to juggle columns: an entropy-based approach for table compression. In: IDEAS, pp. 205–215 (2010)

Final edited form was published in "Performance Evaluation and Benchmarking: Traditional to Big Data to Internet of Things: 7th TPC Technology Conference. Kohala Coast 2015", S. 77–93, ISBN 978-3-319-31409-9
http://dx.doi.org/10.1007/978-3-319-31409-9_6

12. Plattner, H.: A common database approach for OLTP and OLAP using an in-memory column database. In: SIGMOD, pp. 1–2 (2009)
13. Roth, M.A., Van Horn, S.J.: Database compression. SIGMOD Rec. **22**(3), 31–39 (1993)
14. Schlegel, B., Gemulla, R., Lehner, W.: Fast integer compression using SIMD instructions. In: DaMoN Workshop, pp. 34–40 (2010)
15. Stepanov, A.A., Gangolli, A.R., Rose, D.E., Ernst, R.J., Oberoi, P.S.: SIMD-based decoding of posting lists. In: CIKM, pp. 317–326 (2011)
16. Willhalm, T., Popovici, N., Boshmaf, Y., Plattner, H., Zeier, A., Schaffner, J.: SIMD-scan: ultra fast in-memory table scan using on-chip vector processing units. Proc. VLDB Endow. **2**(1), 385–394 (2009)
17. Witten, I.H., Neal, R.M., Cleary, J.G.: Arithmetic coding for data compression. Commun. ACM **30**(6), 520–540 (1987)
18. Zukowski, M., Heman, S., Nes, N., Boncz, P.: Super-scalar RAM-CPU cache compression. In: ICDE, pp. 59–70 (2006)