Direct Transformation Techniques for Compressed Data: General Approach and Application Scenarios

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 techniques like dictionary or run-length compression play an important role in main memory database systems. Having decided for a compression scheme for a dataset, the transformation to another scheme is very inefficient today. The common approach works as follows: First, the compressed data is decompressed using the source decompression algorithm resulting in the materialization of the raw data in main memory. Second, the compression algorithm of the destination scheme is applied. This indirect way relies on existing algorithms, but is very inefficient, since the whole uncompressed data has to be materialized as an intermediate step. To overcome these drawbacks, we propose a novel approach called *direct transformation*, which avoids the materialization of the whole uncompressed data. Our techniques are cache optimized to reduce necessary data accesses. Moreover, we present application scenarios, where such direct transformations can be efficiently applied.

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

1 Introduction

As a consequence, e.g., of the developments in the main memory domain, modern database systems are very often in the position to store their entire data in main memory. Aside from increased main memory capacities, a further driver for in-memory database systems was the shift to a column-oriented storage format in combination with compression techniques. Using both mentioned software concepts, large datasets can be held in main memory with a low memory footprint. That means, modern in-memory database systems have to manage and process large compressed datasets. For compression, lightweight compression techniques have been established in this domain [1,3,5,6,9]. These lightweight techniques provide a good compression rate and they are less CPU intensive than heavyweight approaches like Huffman [4]. Examples of lightweight compression techniques are: dictionary compression [1,9], run-length encoding [1,6] and null suppression [1,6]. Moreover, recent research in the field of lightweight compression techniques increases the performance by the utilization of parallelization concepts like SIMD capabilities of modern CPUs [5,7,8].

Based on the availability of various different lightweight compression schemes, the complexity of the physical database design increases. That means, for each column an appropriate compression scheme has to be identified. Abadi et al. [1] have proposed a decision tree to heuristically decide which compression scheme to use for a column. As they have shown [1], the optimal lightweight compression scheme depends on various influencing factors like the number of distinct values, data locality or access pattern. However, these influencing factors usually change over time. To react in an appropriate way on the physical database layer, efficient techniques to transform compressed data from one compression scheme to another are required. To the best of our knowledge, this aspect has not been considered before for lightweightly compressed data. Therefore, this paper primarily focuses on this aspect.

A naïve transformation approach would be the indirect way from a source to a destination compression scheme. First, the compressed data is decompressed using the source decompression algorithm resulting in the materialization of the raw data in main memory. Second, the compression algorithm of the destination scheme is applied. This indirect way relies on existing algorithms and can be realized for arbitrary pairs of source and destination compression schemes. However, the naïve approach is very inefficient and the whole uncompressed data has to be materialized as an intermediate step. To overcome these drawbacks, we contribute a novel *direct transformation* approach in this paper:

- Our novel *direct transformation* techniques convert compressed data in scheme X to another compression scheme Y in a direct and interleaved way.
- We avoid the materialization of the whole uncompressed data as in the naïve approach. Furthermore, our direct techniques are cache optimized to reduce necessary memory accesses.
- We introduce different direct transformation algorithms in detail.
- In our evaluation, we show that our direct transformation techniques outperform the indirect, classical way to convert the compression scheme.
- Furthermore, we present different application scenarios for our direct transformation techniques.

The remainder of the paper is organized as follows: The next section briefly reviews related work in the context of lightweight compression techniques. In Sect. 3, we describe transformation approaches in general. Then, we present different examples of direct transformation techniques in Sect. 4. Section 5 shows the results of our empirical evaluation. Before we conclude the paper in Sect. 7, we highlight different applications requiring efficient transformation techniques.

2 Related Work

The field of lightweight compression has been studied for decades. The main archetypes of lightweight compression techniques are dictionary compression

(DICT) [1,9], delta coding (DELTA) [5,6], frame-of-reference (FOR) [3,9], null suppression (NS) [1,6], and run-length encoding (RLE) [1,6]. DICT replaces each value by its unique key. DELTA and FOR represent each value as the difference to its predecessor respectively a certain reference value. These three well-known techniques try to represent the original data as a sequence of small integers, which is then suited for actual compression using a scheme from the family of NS. NS is the most well-studied kind of lightweight compression. Its basic idea is the omission of leading zeros in small integers. Finally, RLE tackles uninterrupted sequences of occurrences of the same value, so-called runs. In its compressed format, each run is represented by its value and length, i.e., by two uncompressed integers. Therefore, the compressed data is a sequence of such pairs (see Fig. 1).

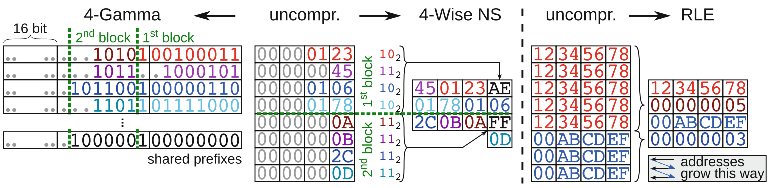

Fig. 1. Examples of some uncompressed data and its representations in the described formats of NS (left) and RLE (right). The data of 4-Gamma is given in binary, whereby gray dots mean leading zero bits. All other data is presented in hexadecimal notation.

In recent years, research in the field of lightweight compression has mainly focussed on the efficient implementation of these schemes on modern hardware. For instance, Zukowski et al. [9] 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, e.g., in [5,7,8]. The techniques 4-Wise Null Suppression and 4-Gamma Coding introduced by Schlegel et al. in [7] are especially important to understand this paper.

4-Wise NS eliminates leading zeros at *byte* level and processes blocks of four values at a time. During compression, the number of leading zero bytes of each of the four values is determined. This yields four 2-bit descriptors, which are combined to an 8-bit compression mask. The compression of the values is done by a SIMD byte permutation bringing the required lower bytes of the values together. This requires a permutation mask being looked up in an offline-created table using the compression mask as a key. After the permutation, the code words have a horizontal layout, i.e., code words of subsequent values are stored in subsequent memory locations. Thus, the compressed data is a sequence of compressed blocks (see Fig. 1). The decompression simply reads the compression mask, looks up the appropriate permutation mask which reinserts the leading zeros bytes and applies the permutation.

4-Gamma eliminates leading zeros at *bit* level and processes blocks of four values at a time. The compression algorithm first determines the minimum number of bits required to represent the highest of the four values. This number is the shared prefix of the block. All values are represented by that many bits and stored using a vertical layout, i.e., each of the four code words is stored to a separate memory word. This requires shift and logical operations, which are done using vectorized instructions. Finally, the unary representation of the shared prefix is stored to a separate memory location. Again, the compressed data is a sequence of compressed blocks (see Fig. 1). The decompression determines the length of the shared prefix and applies appropriate logical and shift operations to the compressed block in order to extract the original values.

3 Transformation Algorithms in General

The aim of transformation algorithms is to change the compressed format some data is represented in. Therefore, the transformation takes data represented in its *source format* as input and outputs the representation of the data in its *destination format*. Note that this is a lossless process, i.e., after the transformation, the original uncompressed data can still be obtained by applying the decompression algorithm of the destination format. We differentiate between two different types of transformations: *indirect* and *direct*, which are described next. For our novel *direct transformation*, we introduce two variants in Sect. 3.2.

3.1 Indirect Vs. Direct Transformations

For the implementation of a transformation, two different approaches exist: (1) *indirect transformations* and (2) *direct transformations*.

Indirect transformations constitute a naïve approach. First, the compressed input data is decompressed using the decompression algorithm belonging to the source format. In this case, the *entire* uncompressed data is materialized in main memory. Finally, the compression algorithm of the destination format is applied to the uncompressed data in order to obtain the representation of the data in the destination format.

Since indirect transformations rely solely on existing compression and decompression algorithms, they can easily be implemented for arbitrary pairs of source and destination formats. However, they suffer from a major inefficiency: The materialization of the uncompressed data as an intermediate step. This requires a lot of expensive load and store operations. Furthermore, it results in a suboptimal cache utilization: when the uncompressed data is read by the recompression, it is not in the caches anymore.

In order to perform transformations efficiently, we propose to employ *direct transformations*. The decisive criterion for direct transformations is that no uncompressed data is written to main memory. Ideally, all intermediate data of a transformation can reside in CPU registers or at worst in the L1 cache. This allows for high-speed access to these intermediate data. We expect considerable speed ups of direct transformations compared to indirect ones. Our experimental results presented in Sect. 5.1 prove this expectation correct. Direct transformations can, for instance, be accomplished by a tightly interleaved execution of parts of the decompression algorithm of the source format and parts of the compression algorithm of the destination format within the body of a loop iterating over the input data. Thereby, intermediate stores and loads to and from memory can be omitted.

We propose to investigate such direct transformations as a new class of algorithms, which is closely related to compression algorithms. Figure 2 again contrasts the data flows of indirect and direct transformations.

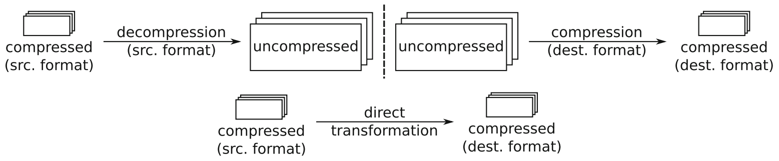
compressed (src. format)
decompression (src. format)
uncompressed
uncompressed
compression (dest. format)
compressed (dest. format)
compressed (src. format)
direct transformation
compressed (dest. format)

Fig. 2. A comparison of the data flows of indirect (top) and direct (bottom) transformations.

3.2 Precise Vs. Imprecise Transformations

In the literature there exists a multitude of compressed formats and associated compression algorithms. Whereas a compressed format specifies the structure of the compressed data, the respective compression algorithm tells how to make use of this structure in order to obtain the best compression rate possible in the given format. Hence, the output of the compression of the destination format could be considered to be a reference for the transformation.

If a transformation algorithm produces a result which equals the result of the compression of its destination format bit by bit, we call it a *precise* transformation. Indirect transformations are always precise, since they actually use the compression algorithm of their destination format. On the other side, direct transformations do not necessarily need to be precise.

For certain combinations of source and destination formats this *bitwise* equality might require a disproportionately high effort. At the same time, data represented in a certain compressed format does not need to use the size reduction potential of the format to its maximum extent. As an example, consider run-length encoding (RLE) [1,6], which replaces each run in the uncompressed data by its value and its length. The uncompressed sequence $[7,7,7,7,7,4,4]$ would be represented as $[(7,5),(4,2)]$ in a precise way. It could, however, also be represented as $[(7,2),(7,3),(4,2)]$ and still be decompressable. Making use of this observation, we introduce a relaxed definition of transformations:

We call a (direct) transformation *imprecise*, if its output O satisfies:

1. O is a valid instance of the destination format.
2. There is some valid input data, such that O is not bitwise equal to the output of the destination format's compression.
3. An application of the decompression of the destination format to O yields uncompressed data that is bitwise identical to the uncompressed data that can be obtained from the output of the respective precise transformation.

The third criterion is especially crucial, since it guarantees that imprecise transformations are in fact *lossless* and do not require any changes to the decompression algorithm. Usually, the result of an imprecise transformation has a bigger size than that of a precise transformation for the same source and destination formats with the same input data. We expect that imprecise transformations might perform better than precise ones for certain pairs of source and destination formats.

In the following section, we present some of our direct transformation algorithms including some imprecise variants.

4 Example Techniques

Figure 3 provides an overview of all transformation techniques, we have investigated so far.[1] However, in this paper we present only a selection of these techniques, namely Rle2FourNs, FourNs2Rle, and FourNs2FourGamma covering all aspects which have to be considered. Currently, we focus on unsigned 32-bit integers as the data type of the uncompressed data. Our algorithms use vectorization through SIMD instructions, since they employ fragments of the vectorized (de)compression algorithms of the involved formats.

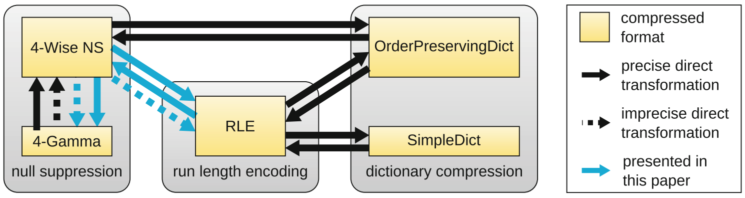

Fig. 3. An overview of the transformation techniques investigated by us so far.

[1] Our source code is downloadable at https://wwwdb.inf.tu-dresden.de/team/staff/patrick-damme-msc/.

4.1 Rle2FourNs

The foundation of the direct transformation from the format of RLE to that of 4-Wise NS is the observation that runs of equal *single values* in the uncompressed data yield (shorter) runs of byte-wise equal homogeneous[2] *compressed blocks* in the compressed format of 4-Wise NS.

The transformation algorithm iterates over its RLE-compressed input data and performs the following steps for each pair of run value and run length:

1. The run value and run length are loaded from the compressed input data.
2. The number of compressed blocks of 4-Wise NS necessary to represent the run is calculated by dividing the run length by four.
3. One block consisting of four copies of the run value is compressed the same way 4-Wise NS would do it. Note that this is done *only once per run*. After this step, the compressed block resides in a vector register in the CPU, i.e., no data is stored to main memory.
4. The compressed block is appended to the output data as often as necessary. This is done by storing the content of the vector register of the previous step to memory multiple times, which does not require any load instructions.

In practice, this procedure gets more complicated, since the run length cannot be assumed to be a multiple of four. In the vicinity of the border between two adjacent runs as well as at the end of the input buffer, it can be necessary to process heterogeneous blocks.

For small run values, this approach can be further accelerated. Storing the compressed block to memory is done using a vectorized store instruction, which writes 16 bytes of vector register content to memory at once. If the run value has exactly one effective byte[3], then the compressed block including the compression mask spans only five bytes. That is, it fits three times into a 16-byte vector register. Hence it is possible to store out three compressed blocks at once. A similar improvement can be made for run values having exactly two effective bytes. In that case, three copies fit into two vector registers. We implemented these optimizations by modifying the permutation masks used by 4-Wise NS to not only permute, but also copy the data within the vector register.

4.2 FourNs2Rle

The direct transformation in the inverse direction, i.e., from the format of 4-Wise NS to the format of RLE, makes use of the fact that runs of byte-wise equal homogeneous *compressed blocks* in the compressed input data mean (longer) runs of equal *single values* in the uncompressed format.

The transformation iterates over all compressed blocks of 4-Wise NS in its input, performing the following steps for each block:

[2] We call a block *homogeneous*, if it contains just one distinct value. Otherwise we call it *heterogeneous*.

[3] Following Schlegel et al. [7], we use the term *effective bits* to denote all but the leading zero bits of a value. The analogous holds for the term *effective bytes*. By definition, the value zero also has one effective bit respectively one effective byte.

1. The compressed block is checked for homogeneity. First, the compression mask is examined. Only if it indicates that all four values have the same length, the actual values are compared *in the compressed form*. If the block is homogeneous, the algorithm continues with step 2, otherwise with step 4.
2. The number of subsequent occurrences of the compressed block is determined *in the compressed input data*, i.e., without decompression. This is done by a simple loop starting at the first byte of the compressed block in the input data. In every iteration, it compares one byte to the corresponding byte in the next block, whose position can be calculated as the block size is known from the compression mask.
3. The one value is extracted from the compressed block and appended to the output as a run value *once*. The run length is obtained by multiplying the number of subsequent occurrences of the compressed block from the previous step by four and appended to the output as well. The algorithm proceeds to the next compressed block and returns to step 1.
4. Since the current compressed block contains more than one distinct value, it is not of interest if it is repeated. Instead, the single block is decompressed to a temporary buffer residing in the L1 cache and immediately recompressed using RLE. The algorithm continues with the next compressed block and returns to step 1.

Hitherto, this yields only an imprecise transformation, which is given the name FourNs2RleImprecise. The reason why the produced output might differ from the output of a direct compression with RLE, is the coarse-grained view on the data. Runs are only determined at block-level, but in fact, 4-Wise NS might partition a run in the uncompressed data into up to three parts: The run might start in a heterogeneous block, run through arbitrarily many homogeneous blocks and finally end in a heterogeneous block again. What FourNs2RleImprecise lacks, is to stitch these parts together. Doing so, however, causes additional overhead. Avoiding this, is the justification for the imprecise technique.

4.3 FourNs2FourGamma

The main idea of the transformation from the format of 4-Wise NS to that of 4-Gamma is a temporary decompression of one compressed block of 4-Wise NS immediately followed by the recompression with 4-Gamma.

The main loop of the algorithm processes each compressed block of 4-Wise NS in the input data as follows:

1. The 8-bit compression mask is loaded and the respective permutation mask for decompression as well as the size of the compressed block are looked up in the tables for the decompression of 4-Wise NS.
2. The decompressing permutation is executed. Note that after this step, the uncompressed block resides in a vector register and does not need to be stored to main memory.

3. The shared prefix of 4-Gamma is determined like in the compression of 4-Gamma, i.e., by computing the maximum number of effective bits via the number of leading zero-bits of the bitwise `OR` of the four values. This is done based on the register contents from the previous step, i.e., without accessing memory. The calculation requires four extractions of a 32-bit element from a vector register, three scalar bitwise `OR` operations, and one invocation of a scalar count-leading-zeros operation.
4. The four values are shifted to right and stored to the values section of the output data, while the shared prefix is stored to the prefix section.

The precise calculation of the maximum number of effective bits of the four uncompressed values in step 3 requires many instructions and therefore costs a lot of time. In order to reduce these costs, the imprecise transformation FourNs2FourGammaImprecise relaxes the strict interpretation of the shared prefix. It approximates the maximum number of effective *bits* of the four values by the maximum number of effective *bytes* increased by eight times. The crucial point is that the latter number can directly be obtained from the 8-bit compression mask of 4-Wise NS by looking it up in a table indexed with the compression mask. This table has a total size of 256 bytes and is created offline. Note that the output data produced this way is perfectly decompressable by 4-Gamma without any changes done to its decompression algorithm. Figure 4 contrasts the result of the precise and the imprecise transformation for an example block.

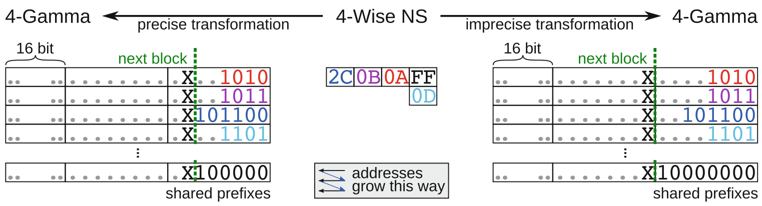

Fig. 4. A comparison of the outputs of the precise and the imprecise variant of the direct transformation FourNs2FourGamma.

5 Experimental Evaluation

We implemented our direct transformation algorithms as well as the corresponding (de)compression algorithms in C++ and compiled them with `g++` 4.8 using the optimization flag `-O3`. Our experiments were conducted on a machine equipped with an Intel Core i7–4710MQ at 2.5 GHz and 16 GB of RAM. The L1 data, L2 and L3 caches have a capacity of 32 KB, 256 KB and 6 MB, respectively.

In all experiments, the underlying uncompressed data consisted of 100 M unsigned 32-bit integers. We use synthetic test data in order to be able to freely specify the properties of the data, especially the distribution of values and the lengths of runs within the data. We report speeds in terms of million integers per second (mis), whereby *integer* refers to an underlying uncompressed value.

5.1 Indirect Vs. Direct Transformations

To find out if direct transformations reach higher speeds than indirect ones, we ran both on the same data and measured the required run times. The (de)compression algorithms employed in the indirect transformations are vectorized and hand-tuned to allow a fair comparison. The results are presented in Fig. 5. The top row of diagrams shows the speeds side by side, while the bottom row explicitly provides the speed ups.

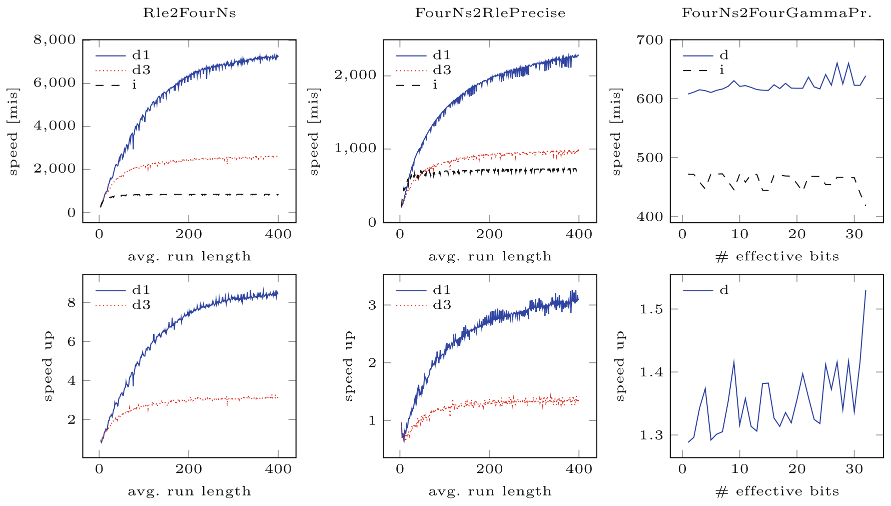

Fig. 5. Comparison of the presented direct transformations (d) to the indirect ones (i).

The first two columns correspond to the transformations involving RLE. Here the data was generated such that it contains runs. The values given at the horizontal axis are the average run lengths. The length of each individual run was chosen uniformly from the interval $avg \pm 2$. We show the results of the direct transformation when all original values have one (d1) or three (d3) effective bytes each, and of the indirect transformation (i), for which the influence of the number of effective bytes would not be visible at the scale of the diagrams.

Both directions of the transformation exhibit the same general trends: (1) the speed increases as the average run length increases, and (2) the more effective bytes the values have, the lower the maximum speed. Except for very small average run lengths, the direct transformations outperform the indirect ones, whereby FourNs2RlePrecise requires run lengths that are a little longer than Rle2FourNs in order to overtake the indirect transformation. The speed ups observed reach up to 8.6 and 3.2 for Rle2FourNs and up to 3.2 and 1.4 for FourNs2RlePrecise, if all values have one respectively three effective bytes.

The third column of Fig. 5 provides the results for FourNs2FourGamma-Precise. In this case, the data was generated such that all values have the same number of effective bits, which is given at the horizontal axis. It can clearly be seen that the direct transformation is faster than the indirect one for all numbers of effective bits. The speed up achieved is between 1.3 and 1.5, which is still considerable.

Our experimental results show that direct transformations are much faster than indirect ones and should thus be employed instead of the latter. We conducted similar experiments for all other direct transformations shown in Fig. 3 and obtained similar results.

5.2 Precise Vs. Imprecise Transformations

In addition to precise transformations, we suggested that it could be faster to perform imprecise transformations in certain cases. We experimentally compared both variants. The results are given in Fig. 6. The top and bottom row of diagrams are concerned with the precise (pr) and imprecise (im) variants of FourNs2Rle and FourNs2FourGamma, respectively. The columns report speeds, speed ups, and compression rates, from left to right.

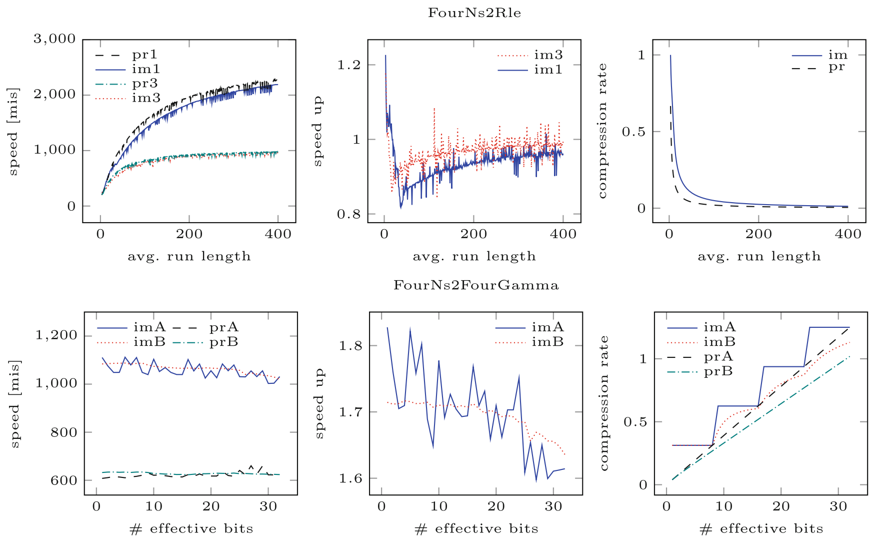

Fig. 6. Comparison of the precise transformations (pr) to the imprecise ones (im).

The results show that the imprecise variant of FourNs2Rle is faster than the precise one only for low average run lengths. A look at the compression rates of the output of the precise and imprecise transformations reveals the reason. As expected, the imprecise variant yields a worse compression rate than the precise one and thus has to store more data. For average run lengths between about 20 and 100, this difference is most significant. For this reason, the precise transformation is clearly faster here. However, this difference in compression rates becomes negligible for long runs. As a consequence, the speed up of the imprecise variant converges on 1.0 again. That is, the imprecise variant yields at least only a slight slow down.

For FourNs2FourGamma we used two different distributions of the original data: (A) all uncompressed values have the same number x of effective bits, and (B) the number of effective bits is chosen uniformly from the interval $[1, x]$, whereby x is the number given at the horizontal axis. In this case, the facts are much clearer. The imprecise variant significantly outperforms its precise counterpart for both distributions and all possible xs yielding speed ups between 1.6 and 1.8, although it leads to a worse compression rate of the output.

To sum it up, the experiments revealed that imprecise direct transformations can indeed be faster then precise ones. However, this is not generally the case as not all combinations of source and destination formats as well as data characteristics seem to be suited. Still, imprecise transformations remain to be an interesting concept and will be promising for other transformation techniques.

6 Application Scenarios

In this section, we stress the usefulness of our direct transformation techniques by presenting two interesting applications requiring efficient transformations.

6.1 Indirect Compression

One possible application of direct transformation techniques is the acceleration of the actual compression. Assume we want to represent some uncompressed data in the compressed format Y. In the classical case, i.e., without considering transformations, there is only one way to achieve this: a *direct compression* by applying the compression algorithm of Y to the uncompressed data. However, also taking transformations into account, there are far more possibilities. We can, for instance, first apply the compression algorithm of some intermediate format X and then perform a transformation from the format X to the format Y. One can trivially see that such an *indirect compression* can only lead to a speed up if it employs a direct transformation. This is due to the fact that an indirect transformation would undo the intermediate compression to the format X as its first step and thus render it to be pure overhead.

The results presented in Fig. 7 prove that such indirect compressions can indeed outperform direct ones. In the example, the compressed format of 4-Wise NS was obtained from uncompressed data by either using the compression algorithm of 4-Wise NS directly (dc) or by the indirection via the format of RLE (ic). Again, the experiment was run for uncompressed values having one or three effective bytes each. The difference of the speed of the direct compression that is subject to the number of effective bytes is negligible at the scale of the diagrams. While the speed of the direct compression is not affected by the average run length, the indirect compression gets faster as the run length increases until it overtakes the direct one at run lengths of about 50 or 150, reaching speed ups of up to 1.8 and 1.3 for one or three effective bytes per uncompressed value.

Unsurprisingly, this does not work for all possible indirect compressions. We conducted the same experiment for the compression to the format of RLE and to the format of 4-Gamma via the format of 4-Wise NS. Both resulted in slow downs compared to the direct compression. Nevertheless, indirect compressions still remain an interesting approach.

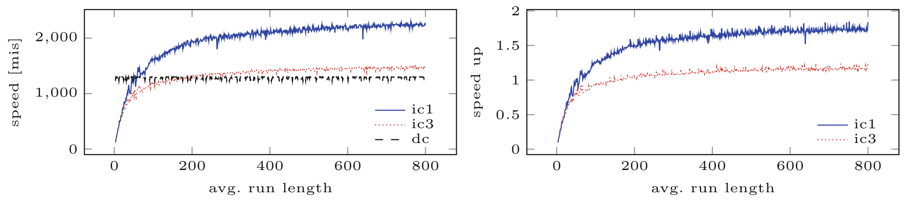

Fig. 7. Comparison of the direct compression (dc) to the format of 4-Wise NS to the indirect one (ic) via the format of RLE.

6.2 Transformations During Query Processing

Another, even more promising application for direct transformations, is the change of the compressed format during query processing. Currently, a shift towards in-memory database systems as the prevailing technology for analytical data processing is taking place. These systems keep *all* their data in main memory, so accessing intermediate results is as expensive as accessing the base data. Thus, intermediate results must be treated efficiently. One way to do so, is to compress not only the base data, but also intermediate results.

Compressed data offers advantages, such as reduced transfer times, better cache utilization, and a higher TLB hit rate. Moreover, many plan operators can directly process compressed data without decompression. On the other side, compression has two major disadvantages. Firstly, it introduces a certain computational overhead, which makes efficient implementations crucial. Recent research [5,7–9] has proven that this is manageable. Secondly, a compressed format has to be chosen. Approaches exist to make this decision wisely, as, e.g. [1], but do not consider the necessity to change the format later.

The optimal format depends on the properties of the data. While the properties of the base data might change only incrementally over time caused by DML operations, the properties of intermediate results usually change dramatically during the processing of a single query. Consequently, operators should be able to output data in another format than their input. For example, a selection might get dictionary-compressed data as input and let only small values pass, such that afterwards a null suppression scheme would be more appropriate. Not adapting the format of the operator's output implies a waste of performance potential. At this point, transformations can be a solution. They could be applied to the output of an operator or even inside an operator. This idea is especially well combinable with the *transient decompression* strategy proposed by Chen et al. in [2]. The authors suggest that operators which are unable to process compressed data directly should temporarily decompress their input, but use the compressed form for the output, again. If a recompression has to be done anyway, it could, of course, provide another format. Note that transformations during query processing must be as efficient as possible, since they are applied online. Due to that, our novel direct transformation techniques are inevitable.

7 Conclusions

In-memory database management systems are of increasing importance in both, business and science. They regularly combine a column-oriented storage format with lightweight compression techniques. The efficient implementation of lightweight compression algorithms as well as the decision for an optimal compressed format have been studied in the literature. However, if the characteristics of the compressed data change over time or during query processing, efficient transformations to other compressed formats can be beneficial, but have not been investigated before. In order to fill this gap, we proposed to use direct transformations that avoid to materialize any uncompressed data in main memory and are cache optimized. Furthermore, we presented precise and imprecise transformations as two variants of lossless direct transformations. Besides a conceptual introduction of such techniques, we also described three concrete algorithms. We conducted an experimental evaluation proving that our new techniques outperform the naïve approach of a complete decompression and recompression. To highlight the usefulness of our direct transformations, we described two possible application scenarios: indirect compression and transformations during query processing.

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. Chen, Z., Gehrke, J., Korn, F.: Query optimization in compressed database systems. SIGMOD Rec. **30**(2), 271–282 (2001)
3. Goldstein, J., Ramakrishnan, R., Shaft, U.: Compressing relations and indexes. In: ICDE, pp. 370–379 (1998)
4. Huffman, D.: A method for the construction of minimum-redundancy codes. Proc. IRE **40**(9), 1098–1101 (1952)
5. Lemire, D., Boytsov, L.: Decoding billions of integers per second through vectorization. In: CoRR abs/1209.2137 (2012)

6. Roth, M.A., Van Horn, S.J.: Database compression. SIGMOD Rec. **22**(3), 31–39 (1993)
7. Schlegel, B., Gemulla, R., Lehner, W.: Fast integer compression using simd instructions. In: DaMoN Workshop, pp. 34–40 (2010)
8. 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)
9. Zukowski, M., Heman, S., Nes, N., Boncz, P.: Super-scalar RAM-CPU cache compression. In: ICDE, pp. 59–70 (2006)