Compression-Aware In-Memory Query Processing: Vision, System Design and Beyond

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

Database Systems Group, Technische Universität Dresden, Dresden, Germany
{juliana.hildebrandt,dirk.habich,patrick.damme,
wolfgang.lehner}@tu-dresden.de
https://wwwdb.inf.tu-dresden.de

Abstract. In-memory database systems have to keep base data as well as intermediate results generated during query processing in main memory. In addition, the effort to access intermediate results is equivalent to the effort to access the base data. Therefore, the optimization of intermediate results is interesting and has a high impact on the performance of the query execution. For this domain, we propose the continuous use of lightweight compression methods for intermediate results and have the aim of developing a balanced query processing approach based on compressed intermediate results. To minimize the overall query execution time, it is important to find a balance between the reduced transfer times and the increased computational effort. This paper provides an overview and presents a system design for our vision. Our system design addresses the challenge of integrating a large and evolving corpus of lightweight data compression algorithms in an in-memory column store. In detail, we present our model-driven approach and describe ongoing research topics to realize our compression-aware query processing vision.

1 Motivation

In-memory database systems pursue a main memory-centric architecture approach and assume that all relevant data can be fully kept in main memory of a computer or of a computer network (cluster configuration) [1,5]. Lightweight data compression methods play an important role in this approach [2,27]. 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. Moreover, this approach is characterized by the fact that all performance-critical operations and internal data structures are designed for efficiently accessing the main memory hierarchy (e.g., efficient use of the cache hierarchy) [14,17]. Furthermore, *any access to an intermediate result generated during query processing is just as expensive as access to the base data* [15,19]. Accordingly, the optimization of the intermediate results is extremely important for an efficient query processing.

S. Blanas et al. (Eds.): ADMS 2016/IMDM 2016, LNCS 10195, pp. 40–56, 2017.
DOI: 10.1007/978-3-319-56111-0_3

1.1 Vision of Compression-Aware In-Memory Query Processing

Generally, two orthogonal techniques are possible to optimize the handling of intermediate results. On the one hand, intermediate results should be no longer produced during query processing. Methods to avoid the generation of intermediate results are (i) adopted code generation for query plans [19] or (ii) the usage of cooperative operators [15]. On the other hand, intermediate results—if they cannot be avoided—should be organized so that an efficient further processing is enabled. In this context, we want to utilize lightweight compression techniques for intermediates as for base data. With the explicit compression of all intermediates, we want

1. to increase the efficiency of individual analytical queries or the throughput of an amount of analytical queries since the main memory requirement is reduced for intermediate results and the extra effort for the generation of the compressed form is minimized, and
2. to establish the continuous handling of compression from the base data to the intermediate results during query processing (holistic approach).

This type of query optimization has been already discussed [6], but not examined in detail since the computational effort for compression and decompression exceeded the benefits of a reduced transfer cost between CPU and main memory. Due to the ever-increasing gap between computing power and main memory bandwidth in modern multiprocessor systems [20] and the recent developments in the domain of efficient lightweight compression methods [18,22,25,27], this argument loses increasingly its validity. Nevertheless, to minimize the overall query execution time, it is important to find a balance between the reduced transfer times and the increased computational effort. To achieve such a balance, not only the query processing but also the necessary part of the query optimization has to be addressed (*compression-aware query processing*).

1.2 System Design Challenge for Compression-Aware Processing

In-memory database systems usually store data according to the decomposition storage model (DSM) [7] to efficiently support analytical and long-running queries. For DSM compression, a large corpus of lightweight data compression algorithms has been developed to efficiently support different data characteristics. Examples are: dictionary compression [2,27], run-length encoding [2,21], and null suppression [2,18,21]. The optimal compression method depends on the properties of the data. If we look at intermediate results, we observe that their properties usually change dramatically during the processing of a single query. Consequently, the compression for intermediate results have to be decided and changed during query processing. 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.

In order to realize our vision, we require an appropriate in-memory system supporting the large corpus of lightweight data compression algorithms. To best

of our knowledge, there is no in-memory database system available providing this large corpus of compression algorithms. Therefore, the most challenging task is now to define a system design allowing us to integrate the large and evolving corpus of data compression algorithms.

1.3 Our Contribution and Outline of the Paper

In this paper, we are primarily focusing on the system design challenge as a fundamental basis for our vision. The naïve approach would be to natively implement the compression algorithms in the DSM storage layer of an in-memory database system as done today. However, this naïve approach has several drawbacks, e.g., (1) massive effort to implement every possible lightweight compression algorithm as well as (2) the integration of new and specific algorithms is time consuming. Therefore, we propose a novel and model-based approach for the integration in this paper. In detail our contributions are:

1. We start with a system design overview in Sect. 2. As we are going to introduce, our solution consists of two components: (i) the unified conceptual model for lightweight compression algorithms and (iii) the transformation of model instances to executable storage layer code.
2. We present our unified conceptual model in detail. We begin with a systematic treatment of the lightweight compression aspect and present a derived system description in Sect. 3. Afterwards, we propose our novel conceptual model *COLLATE* in Sect. 4.
3. We show the applicability of *COLLATE* model by describing two algorithms as model instances in Sect. 5. Then, we highlight our transformation approach to derive efficient executable code out of the model instances.

Furthermore, we close the paper with a description of our ongoing research topics realize our vision of compression-aware in-memory query processing. Finally, we conclude the paper in Sect. 7.

2 System Design Overview

Without loss of generality, we restrict our attention to in-memory column stores, because they are perfectly suited for complex analytical queries from a performance perspective [2,27]. The left side of Fig. 1 shows an abstract architecture of a typical in-memory column-store consisting of three layers: *durability*, *storage*, and *processing layer*. While the *durability layer* guarantees data persistence on non-volatile medium, the *storage* and *processing layer* are the main layers and they are responsible for storing and processing data in main-memory. The *storage layer* itself maintains relational data using the decomposition storage model (DSM) [7]. That means, each attribute is separately stored and the storage equals to a value-based storage model in form of a sequence of values. For the

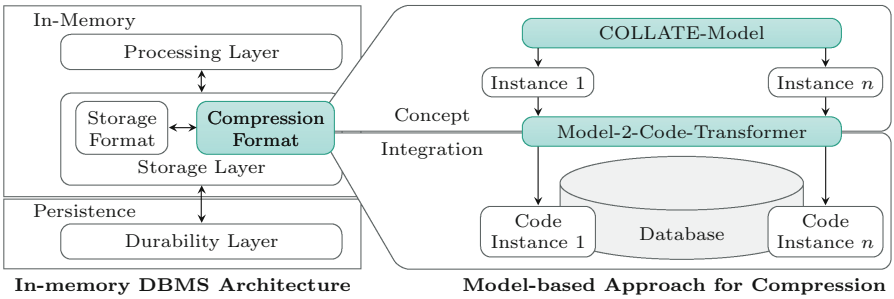

Fig. 1. Model-driven approach for the integration of data compression algorithms.

compression of sequences of values, a large variety of algorithms has been developed [2–4,11,18,21–24,27]. The landscape evolves further because it is impossible to design an algorithm that always produces optimal results for any kind of data.

To avoid the naïve approach by natively implementing each single compression algorithm, we pursue a model-driven approach. Fundamentally, the model-driven architecture (MDA) is a software design approach for the development of software systems [16]. In this approach, the system functionality is defined with a platform-independent model (PIM) using an appropriate domain-specific language [16]. Then, the PIM is automatically translated into one or more platform-specific models (PSM) [16]. The MDA paradigm is widely used in the area of database applications for database creations. On the one hand, the model-driven data modeling and the generation of normalized database schemas should be mentioned. On the other hand, there is the generation of full database applications, including the data schema as well as data layer code, business logic layer code, and even user interface code [12].

In this paper, we propose to use the MDA paradigm for the system-internal domain of lightweight data compression algorithms as illustrated at the right side of Fig. 1. To achieve this, we defined a conceptual model called *COLLATE* for this specific domain. The aim of *COLLATE* is to provide a holistic, abstract and platform-independent view of necessary concepts including all aspects of data, behavior, and interaction. Based on that, a specific compression algorithm can be expressed as model instance. To transform a model instance to executable code, we pursue a generator approach. The generated and optimized code can be used in a column store in a straightforward way.

3 Survey of Lightweight Data Compression Algorithms

Before we present our novel model in the following section, we start with a comparison of basic compression techniques, followed by specific algorithms, and conclude with a system description in this section.

3.1 Analysis of Basic Lightweight Compression Techniques

The basis of lightweight data compression algorithms are six basic techniques: frame-of-reference (FOR) [11,27], delta coding (DELTA) [18,21], dictionary compression (DICT) [2,27], bit vectors (BV) [26], run-length encoding (RLE) [2,21], and null suppression (NS) [2,21]. FOR and DELTA represent each value as the difference to a certain given reference value (FOR) respectively to its predecessor value (DELTA). DICT replaces each value by its unique key given by 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 well-studied kind of lightweight compression techniques. Its basic idea is the omission of leading zeros in the bit representation of small integers. In contrast to DICT, the technique BV replaces each input value with a bit vector representation in the output. 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. Therefore, the compressed data is a sequence of such pairs.

If we analyze these techniques without their application in specific algorithms, we observe the following characteristics:

1. The techniques address different data levels. While FOR, DELTA, DICT, BV, and RLE consider the logical data level, NS addresses the bit or byte level. Therefore, it is clear why algorithms usually combine techniques from the logical level with NS.
2. We are able to distinguish two approaches how input values are mapped to output values. FOR, DELTA, DICT, and BV map each input value to exactly one integer as output value (*1:1 mapping*). The goal is to achieve smaller numbers which can be better compressed on 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*). The NS technique is either a 1:1 mapping or an N:1 mapping depending on the concrete expression.
3. The techniques are either *parameter-dependent* or *data-dependent*. Most of the 1:1 mapping techniques except DELTA are *parameter-dependent*. That means, each input value is independently encoded from other values within the input sequence to an output representation, but the encoding depends on some parameter, i.e., the reference value in FOR or the bit width for NS. DELTA is a *data-dependent technique*, because it encodes the values in dependency of their predecessor. The same is valid for RLE, therefore we define RLE as *data-dependent* technique.

Each technique has its own characteristics and objectives, which are applied in the algorithms in different ways. In particular, the algorithms precisely define some open questions regarding the application. For example, one open question is how the parameter values for *parameter-dependent* techniques are determined. A second open question is whether the whole input sequence is processed with the same parameter value, or if the sequence is partitioned and for each subsequence a separate parameter value is used.

3.2 Analysis of Lightweight Compression Algorithms

Without claiming completeness, we analyzed a large variety of algorithms and classified the algorithms in families as follows:

Algorithms in the family of **Byte-oriented Encodings** rely on the basic technique NS [24]. They map uncompressed values to codewords of a bit length that is a multiple of eight. These algorithms implement NS according to a 1:1 mapping, whereas the corresponding parameter for encoding is determined for each single input value (number of essential bytes). That means, byte-oriented encodings compute the parameters for the *parameter-dependent* NS technique *data-dependent* by computing one parameter per input data value. For decoding, it is necessary to store the length of a codeword as a parameter. The algorithms have a lot of similarities. The only difference between two algorithms is the arrangement of bits and they also differ in the encoding of the length value.

Simple-based Algorithms apply only the NS technique, again [3]. Here, the algorithms try to pack as many binary encoded integer values in a 32 resp. 64 bit codeword by suppressing leading zeros. In contrast to the previous family, the algorithms follow an N:1 mapping for NS. The algorithms subdivide the input sequence in subsequences depending on the size of the input values. In each codeword of a fixed length, several descriptor bits serve as parameters to determine the bit width for all values that are encoded with this codeword. The remaining bits are filled with NS compressed data values. Simple-based algorithms apply the *parameter-dependent* NS technique in a *data-dependent* fashion with one parameter per input data subsequence.

PFOR-based Algorithms implement the FOR technique [27] in combination with NS. They subdivide the input in subsequences of a fix 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.

Adaptive FOR algorithms [10,23] bundle a lot of NS algorithms, but they focus on the problem of how to optimize the subdividing of a finite sequence of integer values in subsequences, such that every value in a subsequence is encoded with the same bit width. These algorithms use the *parameter-dependent* technique NS in a data-dependent fashion with a N:1 mapping approach.

All described families apply one or more basic lightweight compression techniques, but the application is different. For the last three families, data subdivision into subsequences plays an important role. Also the calculation of parameter values that are related to all values within a subsequence like a common bit width is a core part of the algorithms. In general, this aspect influences the encoding of the values in each subsequence differently. That is not addressed on the level of basic techniques, but on the level of the algorithms. Furthermore, the parameters for each subsequence have to be included in the encoded output sequences for decompression purposes. Generally, the *parameter-dependent* basic

techniques are applied in a *data-dependent* fashion in algorithms by computing the parameters based on the input data sequence.

3.3 Derived System Description and Properties

Based on the decomposition storage model [7], the **input** for every lightweight data compression algorithm is a finite *sequence* of (integer) values. The **output** is a sequence of codewords and parameters representing the compressed data. The parameters like bit width or run length are required for decoding/decompression or are part of an access path within the compressed data format. Input and output data have a *logical representation* (semantic level) and a *physical representation* (bit or encoding level). While for some compression techniques it is useful to focus on the semantic level (FOR and DELTA), for other techniques the physical level is more important (NS).

For the transformation from input to output, two further characteristics play an important role for every algorithm. First, most of the basic lightweight compression techniques are *parameter-dependent*. Within the algorithms, a number of parameter values has to be calculated. Second, the basic techniques and algorithms differ in their mapping cardinalities as described above. For *parameter-dependent* N:1 mappings, parameters are calculated for each subsequence of N values. In general, a lot of algorithms *subdivide input data hierarchically* in subsequences (i.e. PFOR) for which the parameters can be calculated. Moreover, the further data processing of a subsequence depends on the subsequence itself. That means, **data subdivision** and **parameter calculation** are important and the application of the basic techniques is then straightforward. Finally, for an exact algorithm description, the **combination** and arrangement of codewords and parameters have to be defined. Here, the algorithms also differ widely.

4 *COLLATE* Model

We now introduce our novel conceptual and platform-independent model called *COLLATE* for the domain of lightweight data compression algorithms. As described in Sect. 3.3, input is a *sequence* of (integer) values and output is a sequence of codewords and parameters representing the compressed data. To convert input data to its compressed output data, several functional concepts with regard to our system description are necessary. Fundamentally, our model consists of the following five main concepts (functional components):

`Recursion`**:** This concept is responsible for the hierarchical data subdivision and for applying the included concepts in the `Recursion` on each data subsequence. Each modeled algorithm is a `Recursion`.

`Tokenizer`**:** This concept is responsible for dividing an input sequence into finite subsequences or single values.

`Parameter Calculator`**:** The concept `Parameter Calculator` determines parameter values for finite subsequences or single values. The specification of the parameter values is done using parameter definitions.

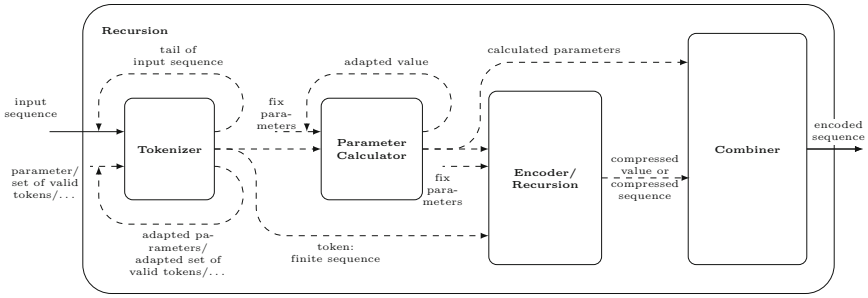

Fig. 2. Interaction and data flow of our *COLLATE* model.

`Encoder:` The third concept determines the encoded form for values to be compressed at bit level. Again, the concrete encoding is specified using functions representing the basic techniques.

`Combiner:` The `Combiner` is essential to arrange the encoded values and the calculated parameters for the output representation.

Figure 2 illustrates the interactions of our concepts and the data flow through the concepts for lightweight data compression for a simple case with only one pair of `Parameter Calculator` and `Encoder`. In general, this arrangement can be used as blueprint for lightweight compression algorithms as we will show in the next section. The dashed lines highlight several properties of the concepts. The properties of the concepts `Tokenizer`, `Parameter Calculator` and `Encoder` are as follows:

For the `Tokenizer` concept, we identified three classifying characteristics. The first one is the *data dependency*. A *data independent* `Tokenizer` outputs a special number of values without regarding the value itself, while a *data dependent* `Tokenizer` is used if the decision how many values to output is led by the knowledge of the concrete values. A second characteristic is the *adaptivity*. A `Tokenizer` is adaptive if the calculation rule changes depending on previously read data. The third property is the *necessary input for decisions*. Most `Tokenizers` need only a finite prefix of a data sequence to decide how many values to output. The rest of the sequence is used as further input for the `Tokenizer` and processed in the same manner. Only those `Tokenizers` are able to process data streams with potentially infinite data sequences. There are also `Tokenizers` needing the whole (finite) input sequence to decide how to subdivide it.

All of these eight combinations are possible. Some of them occur more frequently than others in existing algorithms. Some analyzed algorithms are very complex concerning sequence subdivision. It is not sufficient to assume that `Tokenizers` subdivide sequences in a linear way. As an example for PFOR-based algorithms, we also need `Tokenizers` that arrange somehow subsequences, mostly regarding content of single values of the sequence. With such kinds of `Tokenizers` (mostly categorizable as *non adaptive*, *data dependent* and with the

need of finite input sequences), we can rearrange the values in a different (data dependent) order than the one of the input sequence.

A second task of the `Tokenizer` is to decide for each output sequence which pair of `Parameter Calculator` and `Encoder` is used for the further data processing. Most algorithms process all data in the same way, some of them distinguish several cases, so that this choice is necessary. The finite output sequence of the `Tokenizer` serves as input for the `Parameter Calculator`.

Parameters are often required for the encoding and decoding. Therefore, we introduce the `Parameter Calculator` concept, which knows special rules (parameter definitions) for the calculation of several parameters. There are different kinds of parameter definitions. We often need single numbers like a common bit width for all values or mapping informations for dictionary based encodings. We call a parameter definition *adaptive*, if the knowledge of a calculated parameter for one token (output of the `Tokenizer`) is needed for the calculation of parameters for further tokens at the same hierarchical level. For example, an adaptive parameter definition is necessary for DELTA. Calculated parameters have a logical representation for further calculations and the encoding of values as well as a representation at bit level, because on the one hand they are needed to calculate the encoding of values, on the other hand they have to be stored additionally to allow the decoding. If an algorithm is characterized by hierarchically calculated parameters, it is possible that a parameter definition depends on other calculated parameters that are additional input for a `Parameter Calculator`.

The `Encoder` processes an atomic input, where the output of the `Parameter Calculator` and other parameters are additional inputs. The input is a token that cannot or shall not be subdivided anymore. In practice the `Encoder` mostly gets a single integer value to be mapped into a binary code (1:1 mapping techniques). An exception is RLE as N:1 mapping technique, where the `Parameter Calculator` maps a sequence of equal values to its run length and the `Encoder` maps the sequence to the special value. Equally to the parameter definitions, the `Encoder` calculates a logical representation of its input value and an encoding at bit level.

5 Transformation of Model Instances

In this section, we deal with the application and the transformation of *COLLATE* model instances into executable code, thereby we focus on algorithms from the class of *Byte-oriented Encodings*. Nevertheless, all algorithms belonging to the algorithm families described in Sect. 2 can be modeled with *COLLATE* as demonstrated on our project website[1]. Due to space constraints, we only highlight the basic feasibility of our transformation approach.

5.1 Model Instances for Byte-oriented Encoding Algorithms

The two simple algorithms *varint-SU* and *varint-PU* belong to the class of *Byte-oriented Encodings* and they are designed to suppress leading zeros [24]. Both

[1] *Website* - https://wwwdb.inf.tu-dresden.de/research-projects/projects/collate/.

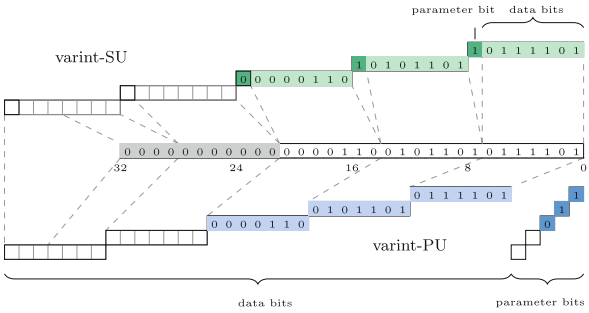

Fig. 3. Example for *varint-SU* (green) and *varint-PU* (blue) (Color figure online).

take 32-bit integer values as input and map them to codewords of variable length. This is shown in Fig. 3 for the binary representation of the value 104,125. Both algorithms determine the smallest number of 7-bit units that are needed for the binary representation of the value without losing any information. In our example, we need at least three 7-bit units and we are able to suppress 11 leading zero bits. To support the decoding, we have to store the number three—number of necessary 7-bit units—as additional parameter. This is done in a unary way as 011. The algorithm *varint-SU* stores each single parameter bit at the high end of a 7-bit data unit, whereas *varint-PU* stores the complete parameter at one end of the 21 data bits.

Both algorithms are similar, which is also observable in their model instances as depicted in Fig. 4. Both algorithms use a very simple `Tokenizer` subdividing an input sequence of length n into single integer values (indicated by $|_{i=1}^{n} m_i$). For each value, the `Parameter Calculator` determines the number of necessary 7-bit units using an appropriate function ($bw = \lfloor \log_{128}(\max(1, \bullet)) \rfloor + 1$). The determined number is used in the subsequent `Encoder`. This one does not transform the logical value, but it subdivides the bit level representation of the input value into bw 7-bit units (indicated by $|_{j=1}^{bw} \bullet_{7j-1} \ldots \bullet_{7(j-1)}$). Up to now, both algorithms have the same behavior. The only difference between both'

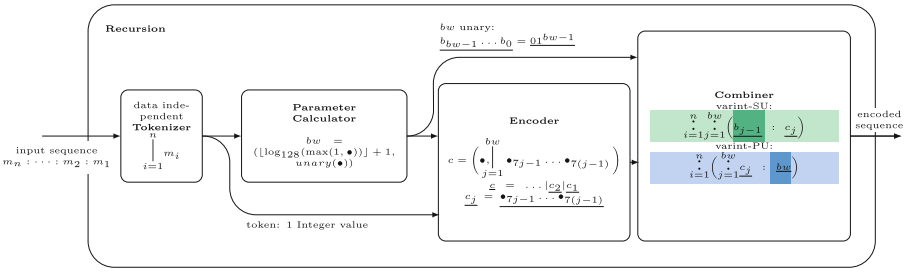

Fig. 4. Model instances for *varint-SU* (green) and *varint-PU* (blue) (Color figure online).

algorithms can be found in the `Combiner`, because only the output is different. The algorithm *varint-SU* concatenates each 7-bit unit of one logical value with one bit of the bit level representation of the parameter *bw* in a loop (indicated by $:_{j=1}^{bw}$) corresponding to the subdivision inside the `Encoder`. It uses a second loop corresponding to the `Tokenizer` to concatenate all *n* values (indicated by $:_{i=1}^{n}$). The algorithm *varint-PU* concatenates the whole bit level representation of a parameter *bw* and all 7-bit units of one value (indicated by $:_{j=1}^{bw}$) as well as all encoded values (indicated by $:_{i=1}^{n}$).

5.2 Transformation to Executable Code

As illustrated above, the algorithms are specified in an abstract way with our novel model approach and appropriate mathematical functions. The next challenge is the transformation in efficient executable code. Figure 5 depicts our developed overall approach for this task, thereby we follow a generator approach. The input of our `Model-2-Code Transformer` are (i) a specification of a model instance in the GNU Octave[2] high-level programming language and (ii) code templates for our model concepts. Our `Model-2-Code Transformer` is written in C and outputs algorithms in C. On the *COLLATE* model level, we have 5 specific concepts and we require one code template for each model concept. The code templates have to be implemented once for each specific database system, e.g., MonetDB [5]. This is necessary to get access to the data on the specific storage layer implementation.

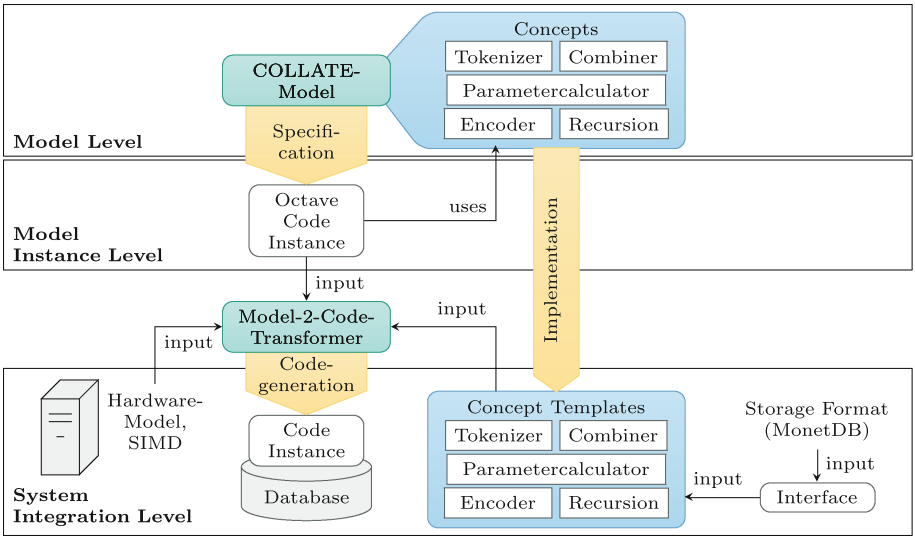

Fig. 5. Transformation of model instances to executable code.

[2] https://www.gnu.org/software/octave/.

The transformation is actually a topic for itself, that is why we want to explain here only the core idea of our approach using the *varint-SU* algorithm as shown in Fig. 4. The `Recursion` concept indicates that we iterate over the input sequence, thereby the `Tokenizer` specifies how we iterate over the input data. That means for our templates, the `Recursion` is a loop and `Tokenizer` information is used to concretize the loop. This approach is shown Fig. 6(a) representing the unoptimized output of our `Model-2-Code Transformer` for *varint-SU*. Furthermore, the `Tokenizer` determines which values have to processed in each loop pass. In our example, we process a single integer value in each loop pass. Then, the concept `Parameter Calculator` includes mathematical rules defining which and how parameters values have to be computed. These definitions are translated in a straightforward way. Generally, `Recursion`, `Tokenizer` and `Parameter Calculator` are working on the logical representation of input values. Afterwards, the `Encoder` gets the calculated parameter values and a single integer value as input. Here, we have to distinguish three opportunities. Either the `Encoder` works on the logical level, on the physical level, or on both levels. On the logical level, the encoding rules are translated again in a straightforward way. If the `Encoder` operates at the physical level, the logical value is usually decomposed in bits or bytes. This is represented in our template using a loop iterating over the bit/byte representation. In our example, the integer value is decomposed in *bw* 7-bit units (see corresponding loop in Fig. 6(a)). This decomposition or bit shifting is extracted from the encoder rule. If the `Encoder` works on the logical as well as physical level, we combine both approaches. Then, the `Combiner` works on the output of the `Encoder` and gets also the parameter as additional input. Again, if the `Combiner` works on the physical level, the corresponding template includes a loop iterating over the bit/byte representation.

To summarize, the templates reflect a foundation and the mathematical rules of the concepts are used to concretize the foundation. Figure 6(a) shows our generated C code we obtained with this approach for *varint-SU*. It is easy to see that this code is not optimal since encoder and combiner contain the same loop. To improve the generated code, we are able to optimize the code using well-known compiler techniques. In this case, we are able to fuse the code for the encoding and combining as illustrated in Fig. 6(b) resulting in one loop iterating over the *bw* 7-bit units. The code can be further optimized by unrolling the loop for encoding and decoding as there are maximal 5 7-bit unit opportunities. That means, we have a specific code part for compression a integer value using one 7-bit unit, specific code part for two 7-bit units, etc. The resulting optimized code is depicted in Fig. 6(c). This optimization is currently executed manually, because the automatically determination of code variants is challenging. This optimization has to be investigated more precisely.

Figure 7(a) shows an evaluation based on single threaded versions of *varint-SU* running on a standard server. We compare our generated and optimized codes to an efficient implementation of Lemire [18]. In this experiment, we vary the number of integer values to be compressed, thereby the compressed representations randomly vary between 1 and 5 7-bit units per integer value.

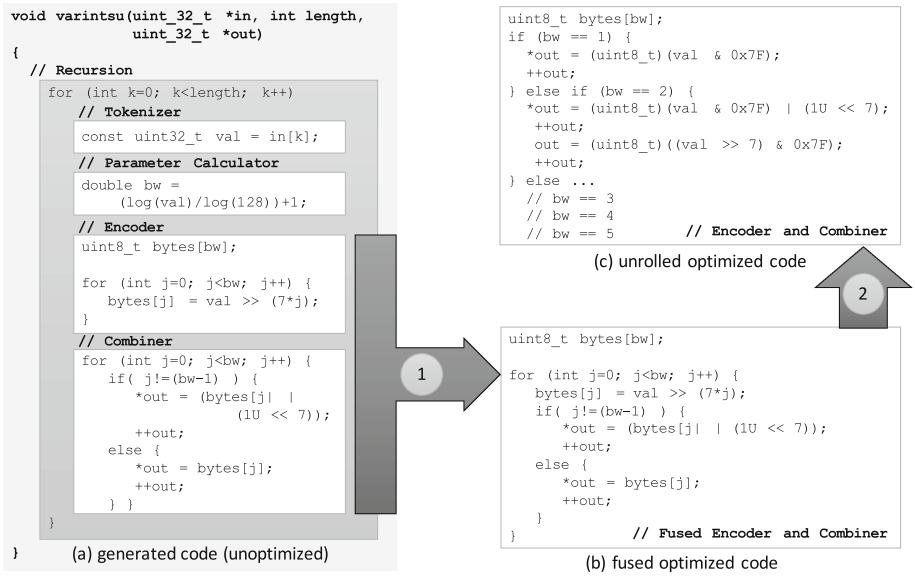

Fig. 6. Generated and optimized code for our *varint-SU* example.

The evaluation is done outside of any column store. As we can see, the unoptimized generated code performs poorly, but with our optimizations, we come close to the native implementation of Lemire. Figure 7(b) indicates the slowdown of our generated and optimized code compared to implementation of Lemire. As we can see, our unoptimized code is around seven times slower than the native code. The fusion of encoder and combiner reduces the slowdown to around 1.5 time, while the unrolling optimization offers the most improvement. Here, our slowdown is around 1.12 compared to the native implementation of Lemire. That means, our generated and optimized code is marginally slower than native code of Lemire.

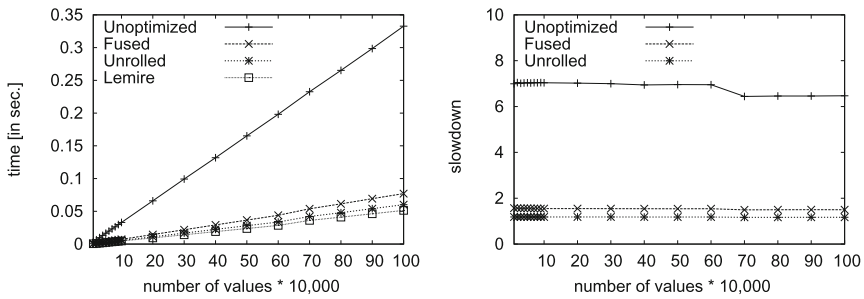

Fig. 7. Evaluation results for *varint-SU*.

In summary it can be established that we are able generate efficient code with our approach. Nevertheless, the whole code optimization has to be worked out in detail, whereby the SIMD-parallelization must be considered with [22,25].

6 Future Work

As mentioned in the introduction, our central aim is to establish a *compression-aware query processing* concept. With the explicit compression of all intermediates, we want to increase the efficiency of individual analytical queries or the throughput of an amount of analytical queries since the main memory requirement is reduced for intermediate results and the extra effort for the generation of the compressed form is minimized. With our presented system design, we have now an appropriate foundation. Generally, our ongoing research activities cover: (i) the integration and optimization of lightweight compression algorithms (structural aspect) in column stores, (ii) the execution of operators on a compressed data format as far as possible (operational aspect), and (iii) the design of an optimization component to decide depending on the situation which compression method should be used for intermediate results (optimization aspect).

6.1 Structural Aspect

With this paper, we have systematically analyzed classical lightweight data compression algorithms. Furthermore, we have proposed a model-driven approach to efficiently integrate the large corpus of algorithms in a column store, e.g., MonetDB [5]. Here, we have focused on the data compression part. The same applies for decompression using a slightly adjusted conceptual model. Our ongoing research topics in this direction are: (i) simplification of algorithm modeling and (ii) extending the transformation/optimization to enable the utilization of SIMD capabilities of modern CPUs. Especially, the second point is interesting because we do not want to parallelize one specific algorithm, but the entire corpus.

A further interesting topic with regard to our vision is the transformation of compressed data in format X into compressed data in format Y. This is important since the optimal compression format depends on the properties of the data [2]. 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, transformation algorithms are our proposed solution as described here [9]. Our direct transformation techniques convert compressed data in format X to another compression format Y in a direct and interleaved way. They could be applied to the output of an operator or even inside an

operator. The transformation algorithms can be also handled with our system design approach.

6.2 Operational Aspect

The operational aspect is another key component for our *compression-aware query processing* aim because physical plan operators have to be designed and implemented, which accept compressed data as an input and provide compressed data as result. The challenge in this task is to ensure that the cost of integrating different combinations of compression formats and operators is as low as possible. There are currently three different strategies for the query execution available which differ in timing and nature of data decompression: *eager decompression* [13], *lazy decompression*[27] and *transient decompression* [6]. In particular, the integration strategy *transient decompression* is very important for our aim. For operators, who can not work on compressed data, the data is decompressed partially and temporarily, however, the compressed representation is used as output. This strategy is intended to form the basis of our approach. However, the fundamental difference is that the intermediate results are transferred in different compression formats in the query plan. This allows changing the optimal compression method in the execution plan, depending on the operators and the data properties.

6.3 Optimization Aspect

At this level, both reduced transfer costs and the overhead of compression and decompression in the search of an optimal execution plan for our compression-aware query processing must be considered. Furthermore, the choice of compression methods for intermediate results and the choice of operator alternatives that can operate on compressed data, are important factors for the query optimization. Our goal is to design a common processing model which includes compression in the query processing as well as optimization. Therefore, further optimization techniques for the compression-sensitive query optimization have to be developed, which can have a major impact on processing times of analytical queries. Our query optimization will be based on a cost model, this cost model has explicit knowledge about the lightweight compression and transformation. This knowledge should be acquired on an empirical evaluation process, thereby we already defined an appropriate benchmark framework [8].

7 Conclusion

In-memory database systems have to keep base data as well as generated intermediate results during query processing in main memory. Furthermore, any access to any intermediate result is just as expensive as access to the base data. Therefore, the intermediate results should be considered separately for an efficient query processing offering two orthogonal optimization approaches:

(i) avoid the generation of intermediate results [15,19] or (ii) organize the intermediate result—if they cannot be avoided—so that an efficient further processing is enabled. In this latter context, we propose to use lightweight compression techniques for intermediates as for base data. In this paper, we explained our overall vision of a compression-aware query processing concept. In particular, we have proposed a model-driven approach to integrate the large and evolving corpus of lightweight data compression algorithms in a column store. Furthermore, we have highlighted our ongoing research activities.

References

1. Abadi, D., Boncz, P.A., Harizopoulos, S., Idreos, S., Madden, S.: The design and implementation of modern column-oriented database systems. Found. Trends Databases **5**(3), 197–280 (2013)
2. Abadi, D.J., Madden, S.R., Ferreira, M.C.: Integrating compression and execution in column-oriented database systems. In: SIGMOD, pp. 671–682 (2006)
3. Anh, V.N., Moffat, A.: Inverted index compression using word-aligned binary codes. Inf. Retr. **8**(1), 151–166 (2005)
4. Arroyuelo, D., González, S., Oyarzún, M., Sepulveda, V.: Document identifier reassignment and run-length-compressed inverted indexes for improved search performance. In: SIGIR, pp. 173–182 (2013)
5. Boncz, P.A., Kersten, M.L., Manegold, S.: Breaking the memory wall in MonetDB. Commun. ACM **51**(12), 77–85 (2008)
6. Chen, Z., Gehrke, J., Korn, F.: Query optimization in compressed database systems. SIGMOD Rec. **30**(2), 271–282 (2001)
7. Copeland, G.P., Khoshafian, S.N.: A decomposition storage model. SIGMOD Rec. **14**(4), 268–279 (1985)
8. Damme, P., Habich, D., Lehner, W.: A benchmark framework for data compression techniques. In: Nambiar, R., Poess, M. (eds.) TPCTC 2015. LNCS, vol. 9508, pp. 77–93. Springer, Cham (2016). doi:10.1007/978-3-319-31409-9_6
9. Damme, P., Habich, D., Lehner, W.: Direct transformation techniques for compressed data: general approach and application scenarios. In: Morzy, T., Valduriez, P., Bellatreche, L. (eds.) ADBIS 2015. LNCS, vol. 9282, pp. 151–165. Springer, Cham (2015). doi:10.1007/978-3-319-23135-8_11
10. Delbru, R., Campinas, S., Samp, K., Tummarello, G., Dangan, L., Delbru, R., Campinas, S., Samp, K., Tummarello, G.: Adaptive frame of reference for compressing inverted lists (2010)
11. Goldstein, J., Ramakrishnan, R., Shaft, U.: Compressing relations and indexes. In: ICDE, pp. 370–379 (1998)
12. Habich, D., Richly, S., Lehner, W.: GignoMDA - exploiting cross-layer optimization for complex database applications. In: VLDB (2006)
13. Iyer, B.R., Wilhite, D.: Data compression support in databases. In: VLDB Conference, pp. 695–704 (1994)
14. Kissinger, T., Schlegel, B., Habich, D., Lehner, W.: KISS-Tree: smart latch-free in-memory indexing on modern architectures. In: DaMoN, pp. 16–23 (2012)
15. Kissinger, T., Schlegel, B., Habich, D., Lehner, W.: QPPT: query processing on prefix trees. In: CIDR 2013 (2013)
16. Kleppe, A., Warmer, J., Bast, W.: MDA Explained. The Model Driven Architecture: Practice and Promise. Addison-Wesley, Massachusetts (2003)

17. Leis, V., Kemper, A., Neumann, T.: The adaptive radix tree: artful indexing for main-memory databases. In: ICDE, pp. 38–49 (2013)
18. Lemire, D., Boytsov, L.: Decoding billions of integers per second through vectorization. Softw. Pract. Exper. **45**(1), 1–29 (2015)
19. Neumann, T.: Efficiently compiling efficient query plans for modern hardware. PVLDB **4**(9), 539–550 (2011)
20. Qiao, L., Raman, V., Reiss, F., Haas, P.J., Lohman, G.M.: Main-memory scan sharing for multi-core cpus. PVLDB **1**, 610–621 (2008)
21. Roth, M.A., Van Horn, S.J.: Database compression. SIGMOD Rec. **22**(3), 31–39 (1993)
22. Schlegel, B., Gemulla, R., Lehner, W.: Fast integer compression using SIMD instructions. In: DaMoN (2010)
23. Silvestri, F., Venturini, R.: Vsencoding: efficient coding and fast decoding of integer lists via dynamic programming. In: CIKM, pp. 1219–1228 (2010)
24. 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)
25. 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. PVLDB **2**(1), 385–394 (2009)
26. Williams, R.: Adaptive Data Compression. Kluwer International Series in Engineering and Computer Science: Communications and Information Theory. Springer, US (1991)
27. Zukowski, M., Heman, S., Nes, N., Boncz, P.: Super-scalar RAM-CPU cache compression. In: ICDE, p. 59 (2006)