Model-Driven Integration of Compression Algorithms in Column-Store Database Systems

Juliana Hildebrandt, Dirk Habich, and Wolfgang Lehner

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

Abstract. 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 decomposition storage model in combination with lightweight data compression algorithms. Using both mentioned storage design concepts, large datasets can be held and processed in main memory with a low memory footprint. In recent years, a large corpus of lightweight data compression algorithms has been developed to efficiently support different data characteristics. In this paper, we present our novel model-driven concept to integrate this large and evolving corpus of lightweight data compression algorithms in column-store database systems. Core components of our concept are (i) a unified conceptual model for lightweight compression algorithms, (ii) specifying algorithms as platform-independent model instances, (iii) transforming model instances into low-level system code, and (iv) integrating low-level system code into a storage layer.

1 Introduction

With an ever increasing volume of data in the era of *Big Data*, the storage requirement for database systems grows quickly. In the same way, the pressure to achieve the required processing performance increases, too. For tackling both aspects in a consistent and uniform way, data compression plays an important role. On the one hand, compression drastically reduces storage requirements. On the other hand, compression also is the cornerstone of an efficient processing capability by enabling in-memory technologies. This aspect is heavily utilized in modern in-memory database systems [1, 23] based on a decomposition storage model (DSM) [7]. For the compression of sequences of integer values like applied in DSM compression or in the compression of posting lists in the context of information retrieval, a large corpus of lightweight data compression algorithms has been developed to efficiently support different data characteristics especially of sequences of integer values. Examples of lightweight compression techniques are: frame-of-reference (FOR) [8, 23], delta coding (DELTA) [15, 19], dictionary compression (DICT) [1, 23], bit vectors (BV) [22], run-length encoding (RLE) [1, 19], and null suppression (NS) [1, 19]. These algorithms achieve good compression rates and they provide fast compression as well as decompression. The

corpus evolves further because it is impossible to design an algorithm that always produces optimal results for all kinds of data. As shown, e.g., in [1, 23, 24], the query performance gain for analytical queries is massive.

From the database system architecture perspective, the most challenging task is now to define an approach allowing us to integrate this large and evolving corpus of lightweight compression algorithms in an efficient way. The naïve approach would be to natively implement the algorithms in the 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) database developers or even users are not able to integrate a specific compression algorithm without a deep system understanding and implementing the algorithm on a system level. Generally, users know their data and could design appropriate compression algorithms, however, database developers have to preserve the generality of the system and cannot implement many algorithms for a small range of applications. To overcome this situation, we propose our novel model-driven approach to easily and automatically integrate almost every lightweight data compression algorithm in an in-memory DSM storage layer. In detail, we address the following points:

1. We start with a brief solution overview in Section 2. As we are going to show, our solution consists of four components: (i) unified conceptual model for lightweight compression algorithms, (ii) description approach for algorithms as model instances, (iii) transforming model instances to executable code and (iv) integration of generated code into the storage layer.
2. Based on this solution overview, we summarize our conceptual model for lightweight compression algorithms in Section 3. This specific conceptual model helps in describing, understanding, communicating, and comparing the algorithms on a conceptual level.
3. Then, we introduce our description language enabling database developers to specify algorithms in a platform-independent form based on our model.
4. These platform-independent model instances are the foundation for database system integration as described in Section 5. Here, we introduce our approach to transform the platform-independent instances to executable algorithms.
5. We conclude with related work and a summary in Section 6 and 7.

2 Solution Overview

Fundamentally, the model-driven architecture (MDA) is a software design approach for the development of software systems [14, 21]. In the MDA approach, the system functionality is defined with a platform-independent model (PIM) using an appropriate domain-specific language [14, 21]. Then, the PIM is translated into one or more platform-specific models (PSMs) that can be executed [14, 21]. 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

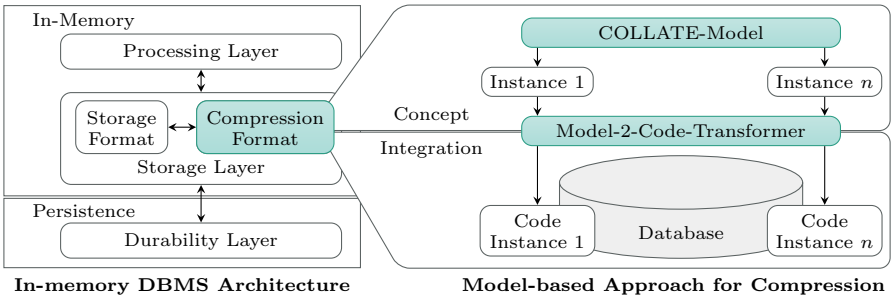


Fig. 1: Model-driven approach for the integration of lightweight data compression algorithms.

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 [10, 18]. Furthermore, the MDA approach has been successfully applied in the area of data warehouse schema creation [17], as well as the modeling and generation of data integration processes [2, 3]. However, there is little to no work on the utilization of MDA for database system-internal components.

As depicted on the left side in Fig. 1, the storage layer of an in-memory database system usually consists of two important components. The first component is the storage format, thereby several formats are proposed. One well-known format is the N-ary storage model (NSM) storing tuples coherently [13]. That means, the tuple unit is preserved in this storage format. The decomposition storage model (DSM), proposed in 1985 [7], is a second widely utilized format. The DSM partitions an n-attribute relation vertically into n sub-relations. Each sub-relation contains two attributes, a logical record id (surrogate) and an attributed value [7]. In most cases, the surrogate can be neglected due to the order of the tuples. That means, sub-relation storage equals to a value-based storage model in form of a sequence of values. While the NSM storage format is used in disk-based database systems, the DSM is the preferred layout of in-memory databases [1, 4, 23, 24]. Compression is the second component of the storage layer, thereby a large and evolving corpus of lightweight data compression algorithms has been developed to efficiently support different data characteristics [1, 15, 23].

For this compression component, we have developed an MDA-based approach as illustrated on the right side in Fig. 1. According to the MDA paradigm, the lightweight data compression algorithms have to be defined in a platform-independent model. To achieve this, we developed a conceptual model for leightweight compression algorithms called *COLLATE* for this specific class of algorithms. The aim of *COLLATE* is to provide a holistic and platform-independent view of necessary concepts including all aspects of data (structure), behavior (function), and interaction (component interaction). In particular, *COLLATE* consists of only five main concepts and we can specify a basic functional arrangement of these concepts as a blueprint for every lightweight data compression algorithm for DSM (see Section 3). That means, the platform-independent

model of a lightweight data compression algorithm is a specific model instance of *COLLATE* expressed in an appropriate language (see Section 4). Then, the platform-specific model can be transformed into a platform-specific executable code as presented in Section 5. The generated code can be used in the database system in a straightforward way. In this paper, we focus on data compression algorithms. The same can be done for the decompression algorithms with a slightly adjusted conceptual model.

3 Conceptual Model

One of our main challenges is the definition of a conceptual model for the class of lightweight data compression algorithms. This is the starting point and anchor of our approach, since all algorithms can be consistently described with this unified and specific model in a platform-independent manner. In [12], we have proposed an appropriate model called *COLLATE* and the development of this model in detail. In the remainder of this section, we briefly summarize its main aspects.

The input for *COLLATE* is a sequence of uncompressed (integer) values due to the DSM storage format. The output is a sequence of compressed values. Input and output data have a logical representation (semantic level) and a physical representation (bit or encoding level). Through the analysis of the available algorithms, we have identified three important aspects. First, there are only six basic techniques which are used in the algorithms. These basic techniques are parameter-dependent and the parameter values are calculated within the algorithms. Second, a lot of algorithms subdivide the input data hierarchically in subsequences for which the parameters can be calculated. The following data processing of a subsequence depends on the subsequence itself. That means, data subdivision and parameter calculation are the adjustment points and the application of the basic techniques is straightforward. Third, for an exact algorithm description, the combination and arrangement of codewords and parameters have to be defined. Here, the algorithms differ widely.

Based on a systematic algorithm analysis, we defined our conceptual model for this class of algorithms. The *COLLATE* model consists of five main concepts—or building blocks—being required to transform a sequence of uncompressed values to a sequence of compressed values:

`Recursion`: Each model instance includes a `Recursion` per se. This concept is responsible for the hierarchical sequence subdivision and for applying the included concepts in the `Recursion` on each data subsequence.

`Tokenizer`: This concept is responsible for dividing an input sequence into finite subsequences of k values (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.

`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.

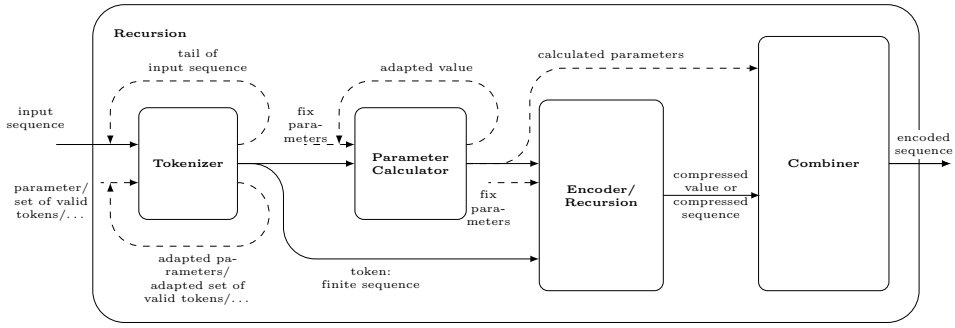


Fig. 2: Interaction and data flow of *COLLATE*.

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

In addition to these individual concepts, Fig. 2 illustrates the interactions and the data flow through our concepts. In this figure, a simple case with only one pair of `Parameter Calculator` and `Encoder` is depicted and can be described as follows. The input data is first processed by a `Tokenizer`. 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 (shown with a dashed line). Moreover, there are `Tokenizers` needing the whole (finite) input sequence to decide how to subdivide it. 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 processing. Most algorithms process all data in the same way, so we need only one pair of `Parameter Calculator` and `Encoder`. Some of them distinguish several cases, so that this choice between several pairs is necessary. The finite `Tokenizer` output sequences serve as input for the `Parameter Calculator` and the `Encoder`.

Parameters are often required for the encoding and decoding. Therefore, we defined the `Parameter Calculator` concept, which knows special rules (parameter definitions) for the calculation of several parameters. Parameters can be used to store a state during data processing. This is depicted with a dashed line. 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.

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. Similar to the parameter definitions, the `Encoder` calculates a logical representation of its input value and an encoding at bit level using functions. Finally, the `Combiner` arranges the encoded values and the calculated parameters for the output representation.

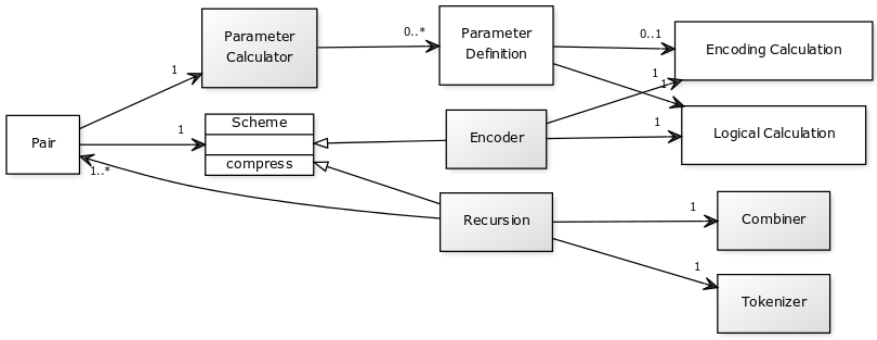


Fig. 3: Class diagram of the *COLLATE* model and data types in the domain-specific language Octave

4 Description Language for Model Instances

Based on our conceptual model, we are able to specify lightweight data compression algorithms in a platform-independent way [12]. For the specification and our overall MDA-based solution, we further require an appropriate language approach. While we introduce our language concept in Section 4.1 in general, we describe an example in Section 4.2.

4.1 Language Approach

Instead of developing a completely new language, we decided to use the GNU Octave[1] high-level programming language as a foundation for the functional behavior of *COLLATE*. A *COLLATE* instance is always a `Recursion` concept. A `Recursion` consists of (1) a `Tokenizer`, (2) a set of pairs of `Parameter Calculator` and a `Scheme`, and (3) a `Combiner` concept. A `Scheme` is either an `Encoder` or a `Recursion`. This relationship can be summarized using a set-oriented notation as follows:

$$\begin{aligned} &ModelInstance \in Recursion, \\ &Recursion = Tokenizer \times \mathcal{P}^{Pair} \times Combiner, \\ &Pair = ParameterCalculator \times Scheme, \\ &Scheme = Recursion \cup Encoder. \end{aligned} \tag{1}$$

Fig. 3 depicts this organization of the *COLLATE* concept as a class diagram in more detail. As described in the previous section, each `Parameter Calculator` can contain several parameter definitions. A parameter definition consists of a logical mapping to calculate a semantic parameter for a subsequence and a bit level mapping (physical level) to calculate the encoding for the semantic parameter in case the parameter has to be encoded. Likewise an `Encoder` consists of a logical mapping and a physical mapping.

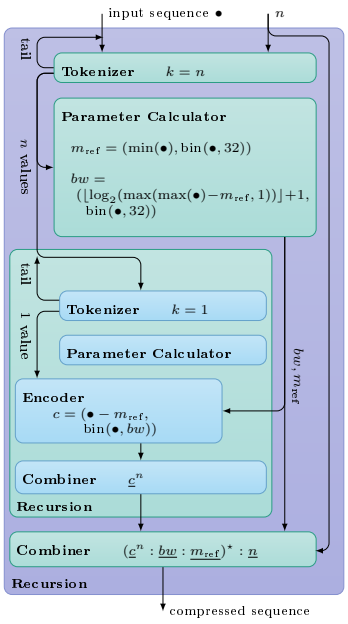
[1] https://www.gnu.org/software/octave/

As mentioned in the Section 3, concepts contain functions for data processing. Therefore, we combine functional and object-oriented programming for the specification of an algorithm in order to preserve the component interaction. According to the class diagram of Fig. 3 and the set-oriented notation, we are able to specify a *COLLATE* model instance as an object of the class `Recursion` in the Octave high level programming language. The `Recursion` class contains (i) a `Tokenizer` object which is characterized by a function handle, (ii) an array of pairs and (iii) a `Combiner` object which is also defined by a function handle. A function handle is an anonymous function with the syntax `@(argument-list) expression`. The input for a `Tokenizer`'s function handle is (1) a sequence `inp` of values and (2) a structure `par`. This structure contains one field resp. attribute for each valid parameter. These attributes might be global parameters that are valid for each subsequence or each single value, or other calculated parameters. The output of a `Tokenizer`'s function handle returns a triple of values. The first element is the number of output values, the second element is a finite subsequence of the input sequence and the third element is a number that indicates, which of the pairs of `Parameter Calculator` and `Scheme` is chosen for the further data processing. A `Combiner`'s function handle has two input values. The first one is an array `inp` of tuples of a compressed sequence or a compressed value `inp.enc` which is the output of a `Scheme` and the corresponding output of the `Parameter Calculator`, `inp.par`. The second one is the structure `par` that contains all other parameters that are valid for the whole array `inp`. The output of a `Combiner` is a concatenation of compressed sequences or values.

Each pair consists of a `Parameter Calculator` object which is defined by several parameter definition objects and an `Encoder` resp. a `Recursion` object. Each parameter definition contains a function handle for the calculation of logical parameters and a function handle for the calculation of the encoding of the logical parameter. The input for the logical calculation is (1) the subsequence `inp` that is an output of the `Tokenizer` and (2) a structure `par` of known and valid parameters. The output is a logical parameter. Its type is not fix. Often it is one single value, but it can be a more complex parameter, i.e., a mapping like a dictionary. The input of the physical calculation is (1) the logical parameter `inp` and (2) all other known and valid parameters `par`. Its output is the encoding for the parameter. The function composition of both function handles maps a finite subsequence to a bit level representation of a parameter that is valid for the sequence `inp` and the parameters `par`. It is the same for the logical and physical function handles of the encoder.

4.2 Example algorithm

To illustrate our Octave language approach, we now present a simple example lightweight data compression algorithm: *frame-of-reference with binary packing for n values (forbp)*. Here, an input sequence of arbitrary length is subdivided in subsequences of n values. The minimum is calculated for each subsequence of n values as the reference value. So, each value of the subsequence can be mapped to its difference to the reference value at the logical data level. This technique is




```
1  CREATE COMPRESSION forbp int AS

2     tok_n = Tokenizer(
          @(inp,par) {par.n.log, [inp(1:par.n.log)], 1});

3     ref = Parameterdefinition("ref",
          @(inp,par) min(inp),
          @(inp,par) dec2bin(inp,32));
4     bw = Parameterdefinition("bw",
          @(inp,par) floor(log2(max(max(inp)-par.ref.log),1))+1,
          @(inp,par) dec2bin(inp,32));
5     pcal_n = ParameterCalculator([ref,bw]);

6     tok_1 = Tokenizer(@(inp,par) {1, [inp(1)], 1});

7     pcal_1 = ParameterCalculator([]);

8     enc_1 = Encoder(
          @(inp,par) inp(1) - par.ref.log,
          @(inp,par) dec2bin(inp,par.bw.log));
9     pair_1 = Pair(pcal_1, enc_1);
10    comb_1 = Combiner(
          @(inp,par) reduce(@(b,c) strcat(c.enc,b),inp,""));
11    rec_n = Recursion(tok_1, [pair_1], comb_1);
12    pair_n = Pair(pcal_n, rec_n);
13    comb_n = Combiner(@(inp,par) reduce(
          @(b,c) strcat(c.enc,c.par.bw.phys,c.par.ref.phys,b),
            inp,par.n.phys));
14    forbp = Recursion(tok_n, [pair_n], comb_n);
```

Fig. 4: Graphical representation and Octave code representation example

called frame-of-reference (FOR) and it is used to get smaller values, which can be encoded with a smaller bit width than 32 resp. 64 bits at the physical data level. The bit level representations of all n deltas can be encoded with a common bit width. Because the deltas are smaller than the original values, a possible common bit width might be smaller than 32 resp. 64 bits. This technique is called binary packing (BP). To guarantee the decodability, the reference value and the used bit width have to be stored additionally to every sequence of n encoded values.

Fig. 4 depicts the graphical representation of the corresponding *COLLATE* instance at the left side and the Octave code representation at the right side. The input ($\bullet$) is an arbitrary sequence of 32-bit integer values. The block size n, a further input, serves as a global parameter to design the algorithm more generic. The whole model instance is an instance of a `Recursion` concept. It consists of a `Tokenizer`, one pair of a `Parameter Calculator` and a `Recursion` and a `Combiner`. The `Tokenizer` outputs the first $k := n$ values. For these n values the `Parameter Calculator` determines the minimum as the reference value $m_{\text{ref}} = \min(\bullet)$ and an appropriate bit width $bw = \lfloor \log_2(\max(\max(\bullet) - m_{\text{ref}}, 1)) \rfloor + 1$ at the logical data level. In the end, both values are encoded in the output with 32 bits each at the physical data level (denoted by the function $\text{bin}(\bullet, 32)$). So each parameter definition consists of one function that addresses the logical data level and one function that considers the encoding resp. physical data level. The next `Recursion` consists of a `Tokenizer`, one pair of `Paramater Calculator` and `Encoder`, and a `Combiner`. This `Tokenizer` outputs single values ($k := 1$). The next `Parameter Calculator` has no task in this instance. Just as parameter

definitions, the `Enocder` consists of one function that processes an input value at the logical level and one function that encodes the value at the physical data level. The `Encoder` determines the difference of the single input value and the reference value at the logical data level (denoted by $c = \bullet - m_{\text{ref}}$) and the encoding of this value with the calculated bit width at the physical data level (denoted by $\text{bin}(\bullet, bw)$). The inner `Combiner` concatenates the physical representations of the n values. This is denoted by $\underline{c}^n$. Physical representations of values are underlined in the graphical model and power n indicates the concatenation of n values. The outer `Combiner` concatenates the n encoded values with the physical representation of their reference value and the physical representation of their bit width for all blocks and the bit level representation of the number n.

The Octave code at the right side contains the same information. In line 2, a `Tokenizer` object is defined. A function handle is input for the constructor. Its return value is the triple consisting of the number n, the first n values of the input sequence and the number resp. address 1. The whole algorithm as the `Recursion` object `forbp` (line 12) is constructed with the `Tokenizer` (line 2), an array with one pair (line 12) of a `Parameter Calculator` (line 5) and `Recursion` (line 11) and a `Combiner` (line 13). The first calculated parameter is the reference value m_{ref}. The parameter definition is constructed in line 3 with one function handle for the logical calculation and a second function handle for the physical calculation. Each concept can be expressed with one or few lines of Octave code.

5 System Integration

The result of the previous two sections is that we are now able to specify lightweight compression algorithms in a platform-independent way by defining model instances with an Octave notation. As depicted in Fig. 4, the first system integration is done using a `CREATE COMPRESSION` statement to register algorithms in the database system under a user-defined name, e.g., `forbp` as in this example. This system integration continues with (i) an approach to transform model instances to executable code and (ii) the specification of the application of the compression algorithm. For this application, we extended the `CREATE TABLE`-syntax in a straightforward way to allow the specification of the compression algorithm which should be used for each attribute separately:

```
CREATE TABLE Test (attribute_a int compress forbp(
  struct("num",struct("log",128,"phys",dec2bin(128,32))));
```

In order to execute model instance specifications inside a database system (e.g. MonetDB [4]), we require an approach to transform model instances to executable code or platform-specific code. Fig. 5 depicts our developed overall approach for this challenge. Generally, we follow a generator approach in our `Model-2-Code Transformer`. At the moment, the input is (i) an Octave specification of a model instance and (ii) code templates for our model concepts. On the *COLLATE* model level, we have 5 specific concepts. That means, we require one code template for each model concept to generate executable code. The code

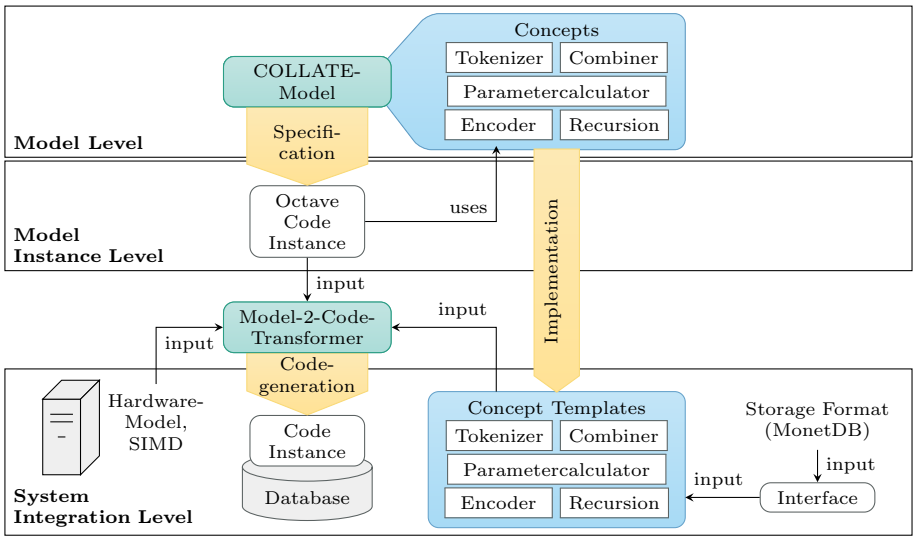


Fig. 5: Transformation of model instances to executable code.

templates have to be implemented once for each specific database system, e.g., MonetDB [4]. This is necessary to get access to data on the specific storage layer implementation.

Based on this input, our `Model-2-Code Transformer` generates the executable code using a replacement and optimization strategy. In this case, the Octave specification is parsed and a corresponding arrangement of the code templates is constructed. The code templates are enriched with the corresponding parameter calculations or encoding functions. Then, the code template arrangement is optimized by applying typical compiler techniques like loop unrolling or load constant replacement [11]. The goal of this optimization is to generate code with less instructions to save as many CPU cycles as possible. This is particularly crucial for reducing the compression overhead.

The generated code can be seamlessly integrated into the specific database system since the templates are implemented for the particular system. At the moment, we are implementing and integrating the whole approach for MonetDB[4], whereas our `Model-2-Code Transformer` is implemented using the LLVM compiler infrastructure. Unfortunately, the current status does not allow a meaningful evaluation with regard to performance.

6 Related Work

In the previous sections, we explained our overall approach for the integration of different data compression algorithms in a column-store database system. With our approach, an in-memory column-store system is extendable with regard to

the bit-level data representation on storage layer level. Generally, extensibility of DBMS was a research field in the late 80's and early 90's [5, 6, 20]. Here, the focus was the integration of new user defined data types, storage methods and indexes, but not the extensibility with regard to new algorithms on top of storage formats. Furthermore, there is also work available focussing on code generation in database systems. For example, Neumann has proposed a completely new query processing architecture [16]. Here, a code generator is used to build specialized query execution plans on demand based on operator templates. He also used the LLVM framework to merge operations into one machine code function. In our work, we follow a similar approach for lightweight data compression algorithms. An interesting research direction would be to combine both approaches for a compression-aware query processing [9].

7 Conclusion

The efficient storage and processing of large datasets is a challenge in the era of *Big Data*. To tackle this challenge, data compression plays an important role from a system-level perspective. Aside from drastically reducing the storage requirement, data compression also enables an efficient processing using "in-memory" technologies. In order to do justice to that, we have presented a model-based approach to integrate a large and evolving corpus of lightweight data compression algorithms in column-store database systems like MonetDB [4] in this paper. Generally, our approach consists of four components: (i) unified conceptual model for lightweight compression algorithms, (ii) description approach for algorithms as model instances, (iii) transforming model instances to executable code and (iv) integration of generated code into the storage layer. In this paper, we concentrated on data compression. The same is also possible for data decompression. In our further research activities, we want to shift our focus to different storage formats to cover not only column-store database systems, but also regard other storage formats, for example row-stores, but our overall approach works.

Acknowledgments

This work was partly funded (1) 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) and (2) by the German Federal Ministry of Education and Research (BMBF) in EXPLOIDS project under grant 16KIS0523.

References

1. Abadi, D.J., Madden, S.R., Ferreira, M.C.: Integrating compression and execution in column-oriented database systems. In: In SIGMOD. pp. 671–682 (2006)
2. Böhm, M., Wloka, U., Habich, D., Lehner, W.: Model-driven generation and optimization of complex integration processes. In: ICEIS. pp. 131–136 (2008)

3. Böhm, M., Wloka, U., Habich, D., Lehner, W.: GCIP: exploiting the generation and optimization of integration processes. In: EDBT. pp. 1128–1131 (2009)
4. Boncz, P.A., Kersten, M.L., Manegold, S.: Breaking the memory wall in monetdb. Commun. ACM 51(12), 77–85 (2008)
5. Carey, M., Haas, L.: Extensible database management systems. SIGMOD Rec. 19(4), 54–60 (Dec 1990)
6. Carey, M.J., DeWitt, D.J., Frank, D., Muralikrishna, M., Graefe, G., Richardson, J.E., Shekita, E.J.: The architecture of the exodus extensible dbms. In: OODS. pp. 52–65 (1986)
7. Copeland, G.P., Khoshafian, S.N.: A decomposition storage model. SIGMOD Rec. 14(4), 268–279 (May 1985)
8. Goldstein, J., Ramakrishnan, R., Shaft, U.: Compressing relations and indexes. In: ICDE. pp. 370–379 (1998)
9. Habich, D., Damme, P., Lehner, W.: Optimierung der Anfrageverarbeitung mittels Kompression der Zwischenergebnisse. In: BTW. pp. 259–278 (2015)
10. Habich, D., Richly, S., Lehner, W.: Gignomda - exploiting cross-layer optimization for complex database applications. In: VLDB (2006)
11. Hänsch, C., Kissinger, T., Habich, D., Lehner, W.: Plan operator specialization using reflective compiler techniques. In: BTW. pp. 363–382 (2015)
12. Hildebrandt, J., Habich, D., Damme, P., Lehner, W.: COLLATE - a conceptual model for lightweight data compression algorithms. Tech. rep., Technische Universitaet Dresden, Database Systems Group (2016), `https://goo.gl/SgXm5z`
13. Kim, W., Chou, H.T., Banerjee, J.: Operations and implementation of complex objects. In: ICDE. pp. 626–633 (1987)
14. Kleppe, A., Warmer, J., Bast, W.: MDA Explained. The Model Driven Architecture: Practice and Promise. Addison-Wesley (2003)
15. Lemire, D., Boytsov, L.: Decoding billions of integers per second through vectorization. Softw., Pract. Exper. 45(1), 1–29 (2015)
16. Neumann, T.: Efficiently compiling efficient query plans for modern hardware. Proc. VLDB Endow. 4(9), 539–550 (Jun 2011)
17. OMG: Common Warehouse Metamodel (CWM), Version 1.0 (2001)
18. Richly, S., Habich, D., Lehner, W.: Gignomda - generation of complex database applications. In: Grundlagen von Datenbanken (2006)
19. Roth, M.A., Horn, S.J.V.: Database compression. SIGMOD Record 22(3), 31–39 (1993)
20. Schwarz, P., Chang, W., Freytag, J.C., Lohman, G., McPherson, J., Mohan, C., Pirahesh, H.: Extensibility in the starburst database system. In: OODS. pp. 85–92 (1986)
21. Thomas, D., Barry, B.M.: Model driven development: the case for domain oriented programming. In: OOPSLA (2003)
22. Williams, R.: Adaptive Data Compression. Kluwer international series in engineering and computer science: Communications and information theory, Springer US (1991)
23. Zukowski, M., Heman, S., Nes, N., Boncz, P.: Super-scalar RAM-CPU cache compression. In: ICDE. p. 59 (2006)
24. Zukowski, M., Nes, N., Boncz, P.A.: DSM vs. NSM: CPU performance tradeoffs in block-oriented query processing. In: DaMoN. pp. 47–54 (2008)