Modularization of Lightweight Data Compression Algorithms

Juliana Hildebrandt, Dirk Habich, Patrick Damme and Wolfgang Lehner

Technische Universität Dresden
Database Systems Group
firstname.lastname@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 column-oriented storage format in combination with lightweight data compression techniques. Using both mentioned software concepts, large datasets can be held and processed in main memory with a low memory footprint. In recent years, a lot of lightweight data compression algorithms have been developed to efficiently support different data characteristics. In our research work, we have investigated a large number of those algorithms to determine similarities and differences between the algorithms. Based on this survey, we developed a novel modularization concept which can be used to describe and to implement a wide range of lightweight compression algorithms in a unified way. In this paper, we introduce our modularization concept and present our model kit implementation. Moreover, we highlight the applicability of our approach using the transcription of different algorithms.

1 Introduction

Data management is a core service for every business or scientific application. The data life cycle comprises different phases starting from understanding external data sources and integrating data into a common database schema. The life cycle continues with an exploitation phase by answering queries against a potentially very large database and closes with archiving activities to store data with respect to legal requirements and cost efficiency. While understanding the data and creating a common database schema is a challenging task from a modeling perspective, efficiently and flexibly storing and processing large datasets is the core requirement from a system architectural perspective [1, 2].

With an ever increasing amount of data in almost all application domains, the storage requirements for database systems grows quickly. In the same way, the pressure to achieve the required processing performance increases, too. To tackle both aspects in a consistent uniform way, data compression plays an important role. On the one hand, data compression drastically reduces storage requirements. On the other hand, compression also is the cornerstone of an efficient processing capability by enabling “in-memory” technologies. As shown in different papers, the performance gain of in-memory data processing is massive because the operations benefit from its higher bandwidth and lower latency [3, 4].

In the area of conventional data compression a multitude of approaches exists. Classic compression techniques like arithmetic coding [5], Huffman [6], or Lempel-Ziv
achieve high compression rates, but the computational effort is high. Therefore, those techniques are usually denoted as heavyweight. Especially for the use in in-memory database systems, a variety of lightweight compression algorithms was developed. These algorithms achieve good compression rates similar to the heavyweight methods by utilizing the context knowledge, but they require a much faster compression as well as decompression. Domain coding \cite{8}, dictionary based compression (Dict) \cite{9, 10, 11}, order preserving encodings \cite{12}, run length encoding (RLE) \cite{13, 14}, frame of reference (FOR) \cite{15, 16} and different kinds of null suppression \cite{17, 18, 19} are examples for lightweight compression algorithms.

In recent years, a lot of lightweight compression algorithms have been developed to efficiently support different data characteristics. Figure 1 shows an overview of various algorithm families and methods, which often build up on the same basic ideas. As illustrated, the algorithms evolve and the development activities increase over the years. Generally, this multitude of algorithms exists because it is impossible to design an algorithm that automatically produces optimal results for any data being stored in a database system. In order to support and to implement a wide range of these algorithms in a database system, a unified approach for the specification or engineering is desirable.

Our Contribution

To tackle this unification, we introduce our novel compression scheme consisting of a few number of modules in this paper. As we are going to show, our compression scheme is quite suitable to modularize a variety of lightweight data compression algorithms in a systematic manner. That means, our approach offers an efficient and an easy-to-use way to describe, to compare, and to adapt lightweight data compression techniques. Aside from presenting our general compression scheme with all modules, we are also highlighting the applicability using different algorithms. Based on
our modularization concept, we developed a first version of a lightweight data compression model kit, which can be used to implement various algorithms in a unified way.

Outline

The remainder of this paper is organized as follows: In the following section, we briefly give an overview on lightweight data compression algorithms classes and introduce our developed novel compression scheme. Then, we use our compression scheme in Section 3 to transcribe different algorithms from two classes. On the one hand, we show the applicability for two algorithms from the null suppression class. On the other hand, we transcribe a semi-adaptive Frame-of-Reference technique with binary packing. Furthermore, we present our model kit implementation being founded on our compression scheme at the end of Section 3. Finally, we conclude the paper in Section 4.

2 General Lightweight Data Compression Scheme

Before we explain our modularization concept in detail, we give a brief overview of lightweight data compression algorithms in the following section.

2.1 Overview of Lightweight Data Compression Algorithms

The field of lightweight data compression has been studied for decades. The main archetypes or classes of lightweight compression techniques are dictionary compression (DICT) [9,10,11], delta coding (DELTA) [13,20], frame-of-reference (FOR) [15,16], null suppression (NS) [14,17,18,19], and run-length encoding (RLE) [13,14]. 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.

2.2 Modularization Concept - Algorithm Scheme

Fundamentally, a modularization concept for data compression was already developed in the 1980s in the context of data transmission. That scheme subdivides compression methods just in (i) a data model adopting to data already read and (ii) a coder which encodes the incoming data by means of the calculated data model. That modularization is suited for the adaptive compression methods as investigated at that time. But it does not adequately reflect different properties of today’s algorithms.
For example, it does not support a sophisticated segmentation or a multi-level data partitioning as required by today’s lightweight data compression methods.

In order to support these new properties, we developed a novel modularization concept for lightweight data compression algorithms, whereas our compression scheme consists of four main modules as shown in Figure 2. Our scheme is a recursion module for subdividing data sequences several times. For reasons of universal validity and comparability we decided, that the input for a compression algorithm does not have to be finite. The first module in each recursion is a Tokenizer splitting the input sequence in finite subsequences or single values at the finest level of granularity. For that, the Tokenizer can be parameterized with a calculation rule. For the Tokenizer module, we further identified three classification 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 lead by the knowledge of the concrete values. A second characteristic is the adaptivity. A Tokenizer is adaptive if the calculation rule changes depending on already read data. Changes of the calculation rules are optional data flows and displayed with a dashed line. The third property is the necessary input for decisions. Most of the Tokenizers need only a finite beginning of a data sequence to decide how many values to output. The rest of the sequence is used as further input for the Tokenizer, processed in the same manner and therefore displayed as an optional data flow with a dashed line. Only those Tokenizers are able to process data streams with potentially infinite data sequences. Moreover, there are 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.

The finite output sequence of the Tokenizer serves as input for the Parameter Calculator, which is our second module. Parameters are often required for the encoding and decoding. Therefore, we introduce this module, whereas this module knows special rules (parameter definitions) for the calculation of several parameters.
There are different kinds of parameter definitions. We need often 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 of our scheme (highlighted by a dashed line). For example, an adaptive parameter definition is necessary for delta encodings [14, 20].

Our third module as depicted in Figure 2 is the *Encoder*, which can be parameterized with a calculation rule for the processing of an atomic input value, whereas the output of the *Parameter Calculator* is an additional input. Its input is a token that cannot or shall not be subdivided anymore. In practice the *Encoder* gets a single integer value to be mapped into a binary code. The fourth and last module is the *Combiner*. It determines how to arrange the output of *Encoder* together with the output of the *Parameter Calculator*. Generally, these four main modules including the illustrated assembly in Figure 2 are enough to specify a large number of lightweight data compression algorithms as described in the next section. Nevertheless, we have to extend our scheme for some algorithms in the following way:

**Recursive Calls:** In some algorithms, parameters like e.g., a common reference value, have to be calculated for sequences of several values. Those algorithms can be represented with one *Tokenizer* outputting for example sequences of 128 integer values. The *Parameter Calculator* determines the common parameter like the minimum of the 128 values as base value. Instead of invoking the *Encoder* module, we call the complete scheme once more again.

**Switch of Parameter Calculator and Encoder:** There are lightweight compression algorithms encoding the data case sensitively. One might want to encode runs of 1s in a different way than other values. So we need a choice of pairs of *Parameter Calculator* and *Encoder*. We determined that the *Tokenizer* chooses the right pair.

**Output of the Tokenizer:** Some analyzed algorithms are very complex concerning sequence partitioning. It is not enough to assume that *Tokenizers* subdivide sequences in a linear way. We need *Tokenizers* that arrange somehow subsequences, mostly in regard to content of the single values of the sequence. With such kind 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. We need this in particular to represent patched coding algorithms.

### 3 Algorithm Transcription

In this section, we demonstrate how lightweight data compression algorithms are transcribed using our proposed modularization concept, whereas we utilize algorithms form different lightweight data compression classes.
3.1 Null Suppression Algorithms

First, we use two simple algorithms (varint-SU and varint-PU) that are designed to suppress leading zeros [19]. Both take 32-bit integer values as input and map them to codes of variable length. This is shown in Figure 3 for the binary representation of the value 104125. 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. In order to support the decoding, we have to store the number three 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 depending on the architecture at one end of the 21 data bits.

The modularization of varint-SU and varint-PU using our compression scheme is highlighted in Figure 4. We use a very simple **Tokenizer** outputting single integer values. This **Tokenizer** instance can be characterized as *data independent* and *non-adaptive*, whereas only the beginning of the data sequence has to be known. For each
value, the **Parameter Calculator** determines the number of necessary 7-bit units. The corresponding formula is depicted in Figure 4. The determined number is used in the subsequent **Encoder** to compute the number of data bits as a multiple of 7. Up to now, both algorithms varint-SU and varint-PU have the same processing procedure. The only difference between both algorithms can be found in the **Combiner**. The notation \( i=0 \to i=n \) is chosen for a concatenation of bit sequences with a counter from \( i=0 \) to \( i=n \), whereas the subsequent operands are concatenated at the high end (left hand according to the chosen notation). The binary representations for whole compressed integer values are concatenated, too, symbolized by a star.

3.2 Frame-of-Reference with Binary Packing Algorithm

For our second example, we utilize a semi-adaptive Frame-of-Reference (FOR) technique with binary packing [15, 20]. In this algorithm, the data sequence has to be partitioned into subsequences of four values. For each subsequence, the minimum value has to be determined, so that the four values can be encoded as the offset to this minimum afterwards. Then, we have to calculate the smallest valid bit width for the binary representation of the encoded values.

![Diagram](image-url)
The modularized version of this algorithm is shown in Figure 5. First, we need a **Tokenizer** that partitions a data stream in finite subsequences containing only four values. The following **Parameter Calculator** determines the minimum as base value as well as the common bit width for each subsequence. Next, the **Recursion module** is used. In this inner recursion, we have a **Tokenizer** subdividing the finite sequence in single integer values, followed by an empty **Parameter Calculator**. Then, we have an **Encoder** calculating the difference between a single integer value and the base value -as a step at the logical level- and encoding it with the calculated bit width -as a step at the bit level. The last module in the inner recursion is a **Combiner** outputting the concatenated offset values as a sequence. This sequence is the input for the **Combiner** of the outer recursion. It concatenates the calculated base value, the bit width and the encoded sequence.

Figure 6 shows an example for the hierarchical organization of the data processing. The minimum of the first four values 12, 10, 9 and 12 is 9. So the base value is 9. The offsets 3, 1, 0 and 3 can be encoded in a binary way with 2 bits each. The inner combiner concatenates the encoded values to 3:1:0:3, the outer one concatenates each sequence with the common bit width and the base value.

### 3.3 Model Kit Implementation

Based on our modularization scheme, we developed an appropriate model kit on the implementation level. Our defined modules are available as building blocks, which can be parameterized with certain calculation rules as described above. In contrast to the logical description, we have to distinguish between logical and physical rules on the implementation level. In particular, the **Parameter Calculator** and the **Encoder** have to be extended with physical rules to establish necessary low-level bit operations. The building blocks can be orchestrated to data flows, so that complete lightweight data compression algorithms can be realized. In our current version¹, we have implemented various concrete algorithms to show the applicability of our approach. Nevertheless, the performance of our algorithms implementation can not

---
¹Our model kit is implemented in C++ and can be downloaded under [https://wwwdb.inf.tu-dresden.de/team/staff/juliana-hildebrandt/](https://wwwdb.inf.tu-dresden.de/team/staff/juliana-hildebrandt/)
be compared to other independent implementations, as we have not considered any optimization so far.

4 Conclusion

In this paper, we have introduced our novel modularization scheme for lightweight data compression algorithms consisting of four modules, which is suitable to modularize a variety of algorithms in a systematic manner. By the replacement of individual modules or only parameters, different algorithms with the same compression scheme can be represented. Some modules and module groups are always occurring in various algorithms, such as the recursion, which accounts for Binary Packing being found in all PFOR- [16, 20, 21] and Simple-algorithms [22]. From our point of view, the understandability of algorithms is improved by the subdivision into different, independent, and small modules that execute manageable operations. Furthermore, our developed modularization scheme is a well-defined foundation for the abstract consideration of lightweight compression algorithms. We are able to represent certain techniques or other properties of compression algorithms as patterns. Static methods such as varint-SU or varint-PU consist only of a Tokenizer, a Parameter Calculator, Encoder, and Combiner module. Adaptive algorithms have an adaptive Tokenizer, an adaptive Parameter Calculator, or both. Semi-adaptive algorithms are characterized by a Parameter Calculator and a recursion, whereas the output Parameter Calculator is used in the Tokenizer or Encoder within the recursion.

In recent years, research in the field of lightweight compression also focused on the efficient implementation of these algorithms on modern hardware. For instance, Zukowski et al. [16] 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 [19, 20]. In our ongoing research work, we are going to integrate such optimization strategies in our model kit.

References


