Integrating Lightweight Compression Capabilities into Apache Arrow

Juliana Hildebrandt[a], Dirk Habich[b] and Wolfgang Lehner[c]
Technische Universität Dresden, Database Systems Group, Dresden, Germany

Keywords: Columnar Data, Data Formats, Apache Arrow, Lightweight Compression, Integration.

Abstract: With the ongoing shift to a data-driven world in almost all application domains, the management and in particular the analytics of large amounts of data gain in importance. For that reason, a variety of new big data systems has been developed in recent years. Aside from that, a revision of the data organization and formats has been initiated as a foundation for these big data systems. In this context, *Apache Arrow* is a novel cross-language development platform for in-memory data with a standardized language-independent columnar memory format. The data is organized for efficient analytic operations on modern hardware, whereby *Apache Arrow* only supports dictionary encoding as a specific compression approach. However, there exists a large corpus of lightweight compression algorithms for columnar data which helps to reduce the necessary memory space as well as to increase the processing performance. Thus, we present a flexible and language-independent approach integrating lightweight compression algorithms into the *Apache Arrow* framework in this paper. With our so-called $Arrow^{Comp}$ approach, we preserve the unique properties of *Apache Arrow*, but enhance the platform with a large variety of lightweight compression capabilities.

1 INTRODUCTION

With increasingly large amounts of data being collected in numerous application areas ranging from science to industry, the importance of online analytical processing (OLAP) workloads increases (Chaudhuri et al., 2011). The majority of this data can be modeled as structured relational tables, thereby a table is conceptually a two-dimensional structure organized in rows and columns. On that kind of data, OLAP queries typically access a small number of columns, but a high number of rows and are, thus, most efficiently processed using a columnar data storage organization (Boncz et al., 2008; Sridhar, 2017). This organization is characterized by the fact that each column of a table is stored separately as a contiguous sequence of values. In recent years, this storage organization has been increasingly applied by a variety of database systems (Abadi et al., 2013; Boncz et al., 2008; Boncz et al., 2005; Stonebraker et al., 2005) as well as big data systems (Kornacker et al., 2015; Sridhar, 2017) with a special focus on OLAP.

Besides the development of such novel systems, we also observe a fundamental revision of the data organization and formats as a foundation for these systems, especially for big data systems (Vohra, 2016a; Vohra, 2016b). Here, the goal is to organize data optimally for processing and analysis (Vohra, 2016a; Vohra, 2016b). A recent and novel approach in this direction is *Apache Arrow*[1] aiming to be a *standardized language-independent columnar memory format for flat and hierarchical data, organized for efficient analytic operations on modern hardware*. The major advantage of this specific memory format is that it acts as a new high-performance and flexible interface for various big data platforms such as Spark (Zaharia et al., 2010), Calcite (Begoli et al., 2018), Impala (Kornacker et al., 2015), or Pandas (Beazley, 2012). That means, data sets represented in this *Apache Arrow* format can be seamlessly shared between different big data platforms with a zero-copy approach.

Additionally, *Apache Arrow* also aims to enable execution engines to take advantage of the latest SIMD (Single Instruction Multiple Data) operations, provided by almost all modern processors, for a highly vectorized and efficient processing of columnar data (Polychroniou et al., 2015; Polychroniou and Ross, 2019; Ungethüm et al., 2020; Zhou and Ross, 2002). However, the gap between the computing power of the CPUs and main memory bandwidth

[a] https://orcid.org/0000-0001-7198-8552
[b] https://orcid.org/0000-0002-8671-5466
[c] https://orcid.org/0000-0001-8107-2775

[1] *Apache Arrow* - https://arrow.apache.org/

Hildebrandt, J., Habich, D. and Lehner, W.
Integrating Lightweight Compression Capabilities into Apache Arrow.
DOI: 10.5220/0009820100550066
In *Proceedings of the 9th International Conference on Data Science, Technology and Applications (DATA 2020)*, pages 55-66
ISBN: 978-989-758-440-4
Copyright © 2020 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

on modern processors continuously increases, which is now the main bottleneck for efficient data processing (Boncz et al., 2008). From that perspective, the columnar data organization is one step to tackle this gap by reducing the memory accesses to the relevant columns. To further increase the processing performance in particular in combination with a vectorized execution, data compression is a second and necessary step to tackle that gap as already successfully shown in the domain of in-memory column store database or processing systems (Abadi et al., 2006; Abadi et al., 2013; Damme et al., 2020; Habich et al., 2019; Hildebrandt et al., 2016). Here, a large corpus of specific lightweight data compression algorithms for columnar data has been developed (Damme et al., 2019; Hildebrandt et al., 2017; Lemire and Boytsov, 2015) and the application dramatically increased the performance of analytical queries (Abadi et al., 2006; Abadi et al., 2013). Unfortunately, *Apache Arrow* only supports dictionary compression in that direction as one specific lightweight compression algorithm.

Our Contribution and Outline. To overcome the current limitation of *Apache Arrow*, we present $Arrow^{Comp}$ as an enhanced approach in this paper. $Arrow^{Comp}$ builds on *Apache Arrow* and integrates the available large corpus of lightweight compression algorithms for columnar data in a flexible and efficient way. Our $Arrow^{Comp}$ approach is language-independent to preserve this important property of *Apache Arrow*. In detail, our main contributions in this paper are:

1. We give a sufficient introduction into *Apache Arrow* as well as into the domain of lightweight data compression algorithms as a foundation for our work in Section 2.
2. Based on this introduction, we explain our developed $Arrow^{Comp}$ approach integrating the large corpus of lightweight compression algorithms based on a metamodel concept in Section 3. This metamodel concept enables us (i) to realize the language independence and (ii) to facilitate the simple integration of a large number of different lightweight compression algorithms without having to implement each algorithm on its own.
3. As a proof of concept, we implemented an $Arrow^{Comp}$ in C++ as described in 3. Using this prototype, we present selective experimental results in Section 4. As we are going to show, $Arrow^{Comp}$ is able (i) to decrease the memory footprint and (ii) to increase the performance of a typical aggregation function compared to the original *Apache Arrow* framework.

Finally, we close the paper with an overview of related work in Section 5 and a conclusion in Section 6.

orderkey	quantity	shipmode
2789	47	REG AIR
9378	48	SHIP
24519	17	REG AIR
37733	35	SHIP

Figure 1: Running example table.

2 PRELIMINARIES

Before we introduce our $Arrow^{Comp}$ approach, this section provides a brief introduction of the underlying *Apache Arrow* framework including a short overview of lightweight compression algorithms. Finally, we close this section with a clear motivation why the integration of lightweight compression algorithms into *Apache Arrow* makes sense.

2.1 Apache Arrow

Generally, *Apache Arrow* aims to be a cross-language development platform for relational in-memory data. On the one hand, this platform defines a *specification* establishing (i) the physical memory layout respectively format of columns in relational data supporting efficient analytical processing with SIMD operations on up-to-date hardware, (ii) the serialization and interprocess communication for data exchange between different systems containing, for example, the structure of table schemas or static column respectively field characteristics like the availability of a name and a data-type, and (iii) guidelines for the implementation scope of engines processing the columnar data. On the other hand, this platform provides implementations respectively *libraries* meeting the specification in different languages like C, C++, Python, Java, etc. Thus, the cross-language property is achieved through the explicit separation of specification and implementation. Moreover, data represented using *Apache Arrow* can be used in different big data platforms such as Spark (Zaharia et al., 2010), Calcite (Begoli et al., 2018), Impala (Kornacker et al., 2015), or Pandas (Beazley, 2012) with a zero-copy approach.

In the following, we (i) introduce some essential vocabulary in terms of relational data used in the specification, (ii) explain a subset of the physical memory layout specification, and (iii) describe the C++ library serving as a foundation for $Arrow^{Comp}$.

2.1.1 Vocabulary

In principle, the *Apache Arrow* specification distinguishes (table) *schemas* and *Apache Arrow* (data) *ar-*

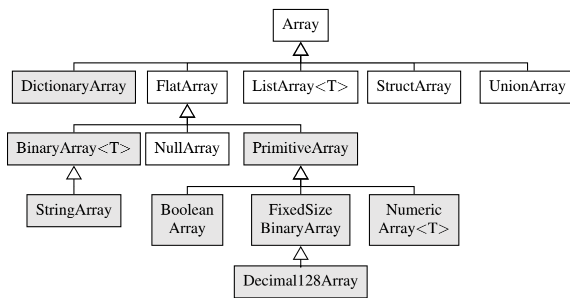


Figure 2: Simplified inheritance hierarchy of of *Apache Arrow* array types.

rays containing the columnar data respectively pieces of columnar data. Schemas consist of an optional table metadata map and a set of *fields*. Fields are static characteristics of table columns, consisting of a name, a data type, a boolean value indicating if null values are allowed, an optional metadata map, and, dependent on the data type, further property declarations. An example table is shown in Figure 1 which we use as a running example. The table schema consists of three fields `orderkey`, `quantity`, and `shipmode`. While the first two fields have the data type `uint32`, the type of the third field is `String`. Moreover, the fields are not `nullable` because all values are required.

2.1.2 Physical Memory Layout of Arrays

Figure 2 depicts a slightly simplified overview of *Apache Arrow* array types for storing columnar data. They can be divided into (i) `FlatArrays` containing non-nested data and (ii) arrays containing nested data types as `ListArrays`, `StructArrays`, and `UnionArrays` with mixed data types. `FlatArrays` can be further divided into `NullArrays` containing exclusively null values, `PrimitiveArrays` with data of the same physical length for each value (i.e. `int32`), and `BinaryArrays` which are variable-size types (i.e. `String`). Moreover, there exists a `DictionaryArray` composed of integral columnar data and a dictionary mapping the original data of arbitrary type to integer values. Beyond this, user-defined extension types can be defined for columnar data, such that for example stored variable binary values can be interpreted in a defined way. In those cases, the field's metadata may contain all further necessary information and parameters for encoding and decoding.

The physical memory layout depends on the column data type and nullability. Columnar data may consist of one or several consecutive memory regions called buffers and primitive characteristic values. Each array has a length. Columns belonging to

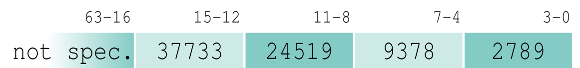


Figure 3: Data buffer with four values padded to 64 bytes.

nullable fields contain a validity bitmap buffer and a null count. `PrimitiveArrays`, `BinaryArrays`, and `DictionaryArrays` (marked in gray in Figure 2) own a single data buffer. *Apache Arrow* arrays with variable size layout contain a further offset buffer. Nested and Union layouts contain one or more data and offset buffers. The internal data buffer structure of nested arrays can be traced back to non-nested array data buffers.

For the efficient analytical processing of a huge amount of data, it is beneficial to store integral (primitive) columnar data as shown in (Abadi et al., 2006; Abadi et al., 2013; Binnig et al., 2009; Müller et al., 2014). For our running example table, this is easy for the fields one and two. The first and second column with `uint32` values would have length 4, null count 0, possibly a validity bitmap buffer, and the data buffer. The values are stored consecutively in a multiple of 64-bit to efficiently support SIMD operations. Thus, padding might be necessary. Figure 3 shows the data buffer for the first column from Figure 1. Bytes 16 to 63 are padded. For the third column, the *Apache Arrow* dictionary encoding can be applied such that the value `REG AIR` is replaced by `0` and the value `SHIP` is replaced by `1`. As a consequence, we store integral values in each column and an additional dictionary in a separate buffer.

2.1.3 C++ Library

The provided C++ library is an implementation of the *Apache Arrow* specification and consists of a stack of 10 different layers. The lowest ones are the `physical layer` with the pure memory management, the one-dimensional layer (`1D Layer` with arrays and the two-dimensional layer (`2D Layer`) implementing schemas, fields, and tables. On top of those, we have seven further layers, e.g., for I/O and interprocess communication (IPC). For this paper, the lower layers are interesting.

In Figure 4, we see our running example table with a schema describing the properties of the three fields and three corresponding columns. Moreover, the implementation in the lower three layers is highlighted. In the C++ library, it is possible to assemble one column from (i) several arrays (chunks) with the same data type or (ii) by one single array as depicted in the *1D Layer* of Figure 4. For each array (chunk), a separate buffer within the `Physical Layer` is allocated. The C++ library of *Apache Arrow* imple-

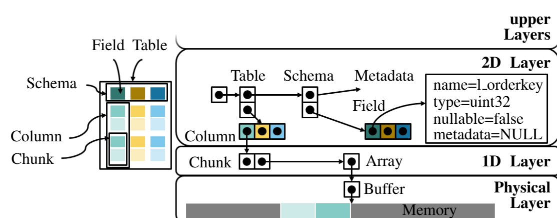


Figure 4: Illustration of the lower three layers of the Apache Arrow C++ library for our running example.

ments the class attributes of the lower layer classes (and nearly everything else) as shared pointers such that often no copies are necessary for data processing.

2.2 Lightweight Integer Compression

Generally, the basic idea of data compression is to invest more computational cycles to reduce the physical data size. This computational effort can be amortized by an increased effective bandwidth of data transfers to storage mediums. This general concept is successfully applied in many areas, among which complex analyses of columnar data are just one example (Abadi et al., 2006; Habich et al., 2019; Hildebrandt et al., 2016; Zukowski et al., 2006). In this area, *lossless* compression is generally preferred.

Depending on the storage medium, different classes of lossless compression can be distinguished. On the one hand, classical *heavyweight algorithms*, such as Huffman (Huffman, 1952), arithmetic coding (Witten et al., 1987), variants of Lempel-Ziv (Ziv and Lempel, 1977; Ziv and Lempel, 1978), and Snappy (Google, 2019) support arbitrary data, but are rather slow. Thus, they are usually employed to amortize disk access latencies. On the other hand, *lightweight compression algorithms*, which have been developed especially for columnar data (Abadi et al., 2006; Hildebrandt et al., 2016; Lemire and Boytsov, 2015), are much faster while still achieving superb compression rates (Damme et al., 2017; Damme et al., 2019). This combination makes them suitable for in-memory columnar processing (Abadi et al., 2006; Damme et al., 2020; Habich et al., 2019).

Lightweight compression algorithms usually focus on integer sequences, which is a natural match for columnar data, since it is state-of-the-art to represent all values as integer keys from a dictionary (Binnig et al., 2009). The unique properties of lightweight compression algorithms result from their exploitation of certain data characteristics, such as the data distribution, sort order, or the number of distinct values in a column (Damme et al., 2017; Damme et al., 2019). Owing to that, a large variety of lightweight integer compression algorithms has been proposed and a recent study showed that there is no single-best one (Damme et al., 2017; Damme et al., 2019).

Examples of lightweight compression include variants of run-length encoding (RLE) (Abadi et al., 2006; Roth and Van Horn, 1993), frame-of-reference (FOR) (Goldstein et al., 1998; Zukowski et al., 2006), differential coding (DELTA) (Lemire and Boytsov, 2015; Roth and Van Horn, 1993), dictionary coding (DICT) (Abadi et al., 2006; Binnig et al., 2009; Roth and Van Horn, 1993; Zukowski et al., 2006), and null suppression (NS) (Abadi et al., 2006; Lemire and Boytsov, 2015; Roth and Van Horn, 1993). To better understand these basic variants, we will briefly explain each of them. FOR and DELTA represent each integer value as the difference to either a certain given reference value (FOR) or to its predecessor value (DELTA). For decompression purposes, we have to store the reference values in FOR. DICT replaces each value by its unique key in a dictionary, whereby this technique can be used to map values of any data type to integer values (Binnig et al., 2009). The dictionary has to be stored for decompression as well. The objective of these three well-known basic variants is to represent the original integer data as a sequence of small integers, which is then suited for actual compression using the NS technique. NS is the most studied lightweight compression technique. Its basic idea is the omission of leading zeros in the bit representation of small integers. Finally, RLE tackles uninterrupted sequences of occurrences of the same value, so-called runs. Each run is represented by its value and length.

2.3 Lessons Learned

To sum up, *Apache Arrow* is an interesting platform as it provides a standardized language-independent columnar in-memory format, organized for efficient analytical operations on modern hardware. It already includes dictionary coding to map arbitrary data types to integer values. However, *Apache Arrow* lacks support for more sophisticated lightweight integer compression algorithms which are suitable (i) to reduce the memory footprint and (ii) to speedup the columnar data processing. To overcome that shortcoming of *Apache Arrow*, we developed an approach to integrate the large corpus of lightweight into *Apache Arrow*.

3 *Arrow*^{Comp} FRAMEWORK

In (Damme et al., 2017; Damme et al., 2019), the authors clarify the large corpus of lightweight integer compression algorithms and experimentally prove

that there is no single-best one but the decision depends on data characteristic and hardware properties. For this reason, it does not make sense to integrate only some lightweight integer compression into *Apache Arrow*, which would only limit the benefits. To achieve the full potential, the complete corpus of algorithms has to be integrated. However, this integration has to be language-independent to preserve this unique property of *Apache Arrow*. To satisfy these challenges, our *Arrow^{Comp}* approach is based on two basic pillars:

1. We already developed a metamodel to specify lightweight integer compression algorithms in a descriptive, language-independent, and non-technical way (Hildebrandt et al., 2017).
2. We evolved the specification of each column to a self-describing compressed column. That means, each column includes a description about the (de)compression as metadata using our metamodel.

To show the feasibility and to evaluate our approach, we implemented a prototype of *Arrow^{Comp}* based on the C++-library of *Apache Arrow*. In the following, we describe each pillar and our implementation in more detail.

3.1 Metamodel for (De)Compression

To specify lightweight integer compression algorithms in a language-independent approach, we already presented an appropriate metamodel in (Hildebrandt et al., 2017). Our metamodel consists of five main concepts processing a stream or batch of values and it defines a data compression format. Each concept contains mainly functions consuming streams or batches and generating other streams, batches, or statistical parameters. The model for each algorithm is a `compression` concept. Each compression contains (1) a `tokenizer`, a function which determines how many consecutive values belong to the next batch, (2) a `parameter calculator`, a set of functions, which determine and encode statistical parameters, (3) an `encoder`, a tuple of functions calculating the encoding of a batch or a further compression model, such that compression models can be nested, and (4) a `combiner` that determines the serialization of encoded batches and parameters.

Figure 5 depicts a concrete example using a simple null suppression algorithm SIMD-BP128 (Lemire and Boytsov, 2015). The basic idea of SIMD-BP128 is to partition a sequence of integer values into batches of 128 integer values and compress every value in a batch using a fixed bit width (namely, the effective bit width of the largest value in the block) (Lemire and Boytsov, 2015). That means, the compression is achieved by omission of leading zeros in the bit representation of each integer value. As shown in Figure 5, the corresponding metamodel instance of SIMD-BP128 includes only this non-technical information. In detail, the first `tokenizer` (line 2) partitions the incoming sequence of integers denoted as *in* into batches with a size of 128 values. The subsequent `parameter calculator` (line 3) determines the bit width of the largest value using a function *in.max.bw* and this bit width is stored in a variable *width*. Then, a new compression part (line 4) starts with a second `tokenizer` (line 5) partitioning the batches in new batches of size one, so that the `encoder` (line 7) can represent each value with the determined bit width (`in.bin(width)`). In this case, we do not need any further parameters and thus, we use an empty `parameter calculator` (line 6). Last but not least, the `combiners` (lines 8 and 10) specify the concatenation of the compressed representation, whereby the used bit width per batch is included in the compressed representation (line 10). This information is necessary for a successful decompression.

```
1  compression(SIMD-BP128,
2    tokenizer(in => 128),
3    parametercalculator(width, in.max.bw),
4    compression(
5      tokenizer(in => 1),
6      parametercalculator(),
7      encoder(in => in.bin (width)),
8      combiner(in => in)
9    )
10   combiner (in => in o width)
11 )
```

Figure 5: SIMD-BP128 metamodel instance.

For *Arrow^{Comp}*, we straightforwardly extended our metamodel to be able to specify the decompression algorithm in the same way as well. Since the specific parameter values for the compression, such as the used bit width per batch, are always included in the compressed data, they only have to be extracted instead of being calculated using a `parameter calculator`. Then, this information is used in the `encoder` to decompress the values.

To sum up, with our metamodel approach, we are able to specify the compression as well as decompression procedure for a large set of lightweight compression algorithms in a descriptive and language-independent manner. Moreover, our metamodel consists of only five specific concepts which are enough for that lightweight integer compression domain.

3.2 Self-describing Column

Since data properties have a significant influence on the choice of the best-fitting lightweight compression algorithm (Damme et al., 2017; Damme et al., 2019), each column has to be considered separately. However, this also means that we have to specify for each column with which algorithm it is compressed and how it has to be decompressed. Moreover, this specification has to be done in an implementation-independent form so that the same format can be utilized by different systems or libraries. To achieve that, we decided to include the metamodel instances for compression and decompression in the corresponding field metadata of the columns in *Apache Arrow*. We borrow this approach from the extension types idea as presented in Section 2.1. Figure 6 illustrates our approach for the first column.

To sum up, our general concept is to store the compressed column together with a description to specify and generate the compression as well as decompression procedure (compressed data together with (de)compression algorithms). From our point of view, this is the most flexible approach to make data interchangeable between different systems. However, this also means that the systems have to be able to interpret and to execute the descriptions of the (de)compression procedures. To achieve that, we propose that the separation of specification and implementation has to be applied for algorithms as done with our metamodel approach.

Furthermore, our self-describing column approach seamlessly interacts with the already available dictionary encoding of *Apache Arrow*. Based on that, can fully utilize the potential of the lightweight integer compression algorithms for all kinds of data types by (i) encode the values of each column as a sequence of integers using dictionary encoding and (ii) apply lightweight lossless integer compression to each sequence of integers resulting in a sequence of compressed column codes.

3.3 Implementation

To prove the feasibility and applicability of our approach, we implemented an *Arrow*^{Comp} prototype based on the C++ library of *Apache Arrow*. For this prototype, the following challenges arose: (i) integrating metamodel instances into metadata, (ii) converting metamodel instances to efficient executable code, (iii) provision of access to the compressed data. In the following, we present our solutions for that as summarized in Figure 6.

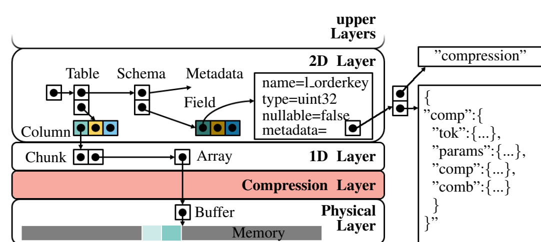


Figure 6: *Arrow*^{Comp} layer structure providing compressed columnar storage and processing capabilities.

3.3.1 Metadata Integration

To integrate the metamodel instances for compression and decompression, we decided to include those descriptions in the corresponding field metadata of the columns. As we already mentioned above, we borrowed this integration concept from the extension types idea from *Apache Arrow*. In our current implementation, we describe the metamodel instances as JSON objects, because the algorithms are usually composite tree structures which perfectly match to JSON.

3.3.2 Converting to Executable Code

To modify and to process columnar data, the metamodel instances of compression and decompression have to be converted to executable code. To achieve that, there are two possible ways: (i) code generation out of the metamodel or (ii) by provision of an implemented library with the required functionality. In general, both ways are feasible, but we decided to implement a library as an additional layer between the `Physical Layer` and the `1D Layer` of *Apache Arrow* as illustrated in Figure 6. The task of this library is to compress and to decompress columnar data according to the descriptions in the corresponding metadata of the column in a transparent way. That means, the `1D Layer` does not know that the columnar data is stored in a compressed way at the `Physical Layer`.

In general, our library is based on C++ template metaprogramming. In more detail, every metamodel concept such as `compression` or `tokenizer` is realized as a template which can be parameterized and tailored using function inlining. Then, these templates are orchestrated as specified in the metamodel instances The advantage of this approach is that we can directly derive a library implementation out of a JSON-represented compression or decompression algorithm using a regular compiler. Since most lightweight integer compression implementations have been proposed for vectorization with vector registers of size 128-bit (Damme et al., 2019; Lemire

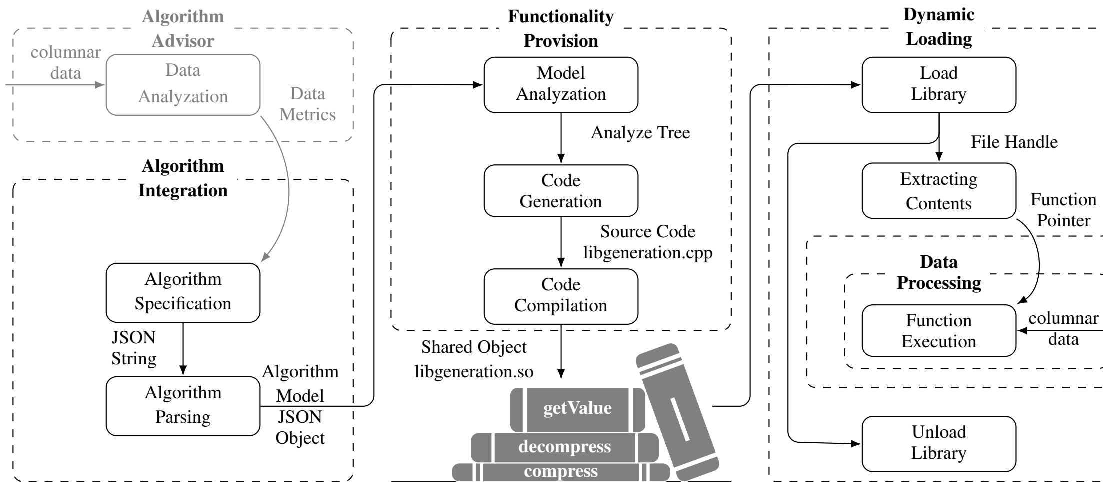


Figure 7: Runtime steps during the life time of a data format object (i) to integrate algorithms, (ii) to provide the algorithmic functionality, and (iii) to load and to execute the generated code.

and Boytsov, 2015), our templates currently support this vector register sizes as well.

Moreover, we introduced a new data format class in *Arrow*^{Comp}. This new data format corresponds to the model instance. Figure 7 summarizes all steps during the lifetime of a data format object. When a data format is instantiated (line 4 in Figure 8), the input JSON Strings specifying the compression and decompression algorithms are parsed to JSON objects. To generate the corresponding executable code, the metamodel descriptions are analyzed in a next step. This includes the calculation of positions and lengths of bit strings, which have to be encoded, the application of case distinctions and loop unrollings as well as the determination of the output granularity like word granularity or byte granularity. Afterward a couple of functions is compiled to a file belonging to the special data format and a shared library is constructed. This library is loaded dynamically at run time, such that the library functions are accessible by a (function) pointer and can be applied for data processing like compression, decompression, as well as value access as discussed as next.

3.3.3 Providing Access

The construction of a compressed array and reading of a sequence of integer values is shown in Figure 8. *Apache Arrow* uses a so-called builder to construct an *Apache Arrow* array. The builder organizes the incremental building of different buffers corresponding to different data types, calculates null counts, etc. To store compressed data in *Arrow*^{Comp}, the builder for a compressed array has to know the compressed data format. This is done by setting the (de)compression JSON-objects for the data format for a specific field

```
1  // setting the table schema
2  vector<shared_ptr<Field>> schema =
       {field("orderkey", uint32()),...};
3  // setting (de)compression infos
4  auto mycomp = arrowcomp::setFormat<UInt32Type>
       (schema->field(0), JSON-String
       Compression, JSON-String Decompression);
5  // building data arrays
6  auto builder = arrowcomp::builder(mycomp);
7  // execution of compression code
8  builder.Append(2789);
9  builder.Append(9378);
10 arrow::Status st = builder.Finish();
11 // reading compressed data
12 auto vals (new arrowcomp::
       CompressedArray<UInt32Type>(builder));
13 // decompression of a single value
14 auto val = vals->Value(1);     // == 9378
```

Figure 8: Modifying and reading compressed arrays.

(line 4). Then, the builder gets this information as input (line 6), whereby the required code for compression and decompression will be initialized. Afterward, we can append single values to the column and these values are transparently compressed using the specific compression algorithm (lines 8-10).

To read the values of a compressed column, we access the values in a regular uncompressed way (line 14). In the background, our introduced `Compression Layer` is responsible for the decompression to return uncompressed values to the application. Important to note, our `Compression Layer` does not decompress the entire column, but only in pieces that fit in the cache hierarchy. In that way, we can efficiently execute *sequential access* on the entire column with decompressing only small parts. To realize random access efficiently, we need to determine which part of

the compressed column has to be decompressed.

In most cases, the compressed data is organized in batches. That means, the number of values in one batch is fix, e.g., 128 values in SIMD-BP128, but the physical data size varies, because a different bit width is used for each block. Here, we can exploit chunked arrays, such that each batch is mapped to exactly one chunk. A second case is, that the compressed data is organized in blocks with a constant physical data size, but the number of values varies. Examples are word-aligned algorithms with a 2- to 4-bit header that encodes the number and bit lengths of the bit-packed values or RLE. Here, an index tree structure can be applied, such that the block which contains the value at a given position can be found in a constant time. Depending on the block size, exploiting chunks or offsets can be advantageous. However, these data formats are more suitable for sequential than for random access.

In a third case, the compressed data is not organized in blocks and all values have the same length. Here random access is trivial. The fourth and last case includes variable-length compressed values, which are not organized in blocks. These data formats are also more suitable for sequential than for random access, but here it would be possible to organize a fixed number of values in chunks. An orthogonal approach to access variable-length data is to store the length information and the data itself separately, such that only the length information has to be iterated to find the value you are looking for. This can be specified with a multi-output combiner in our metamodel.

4 EVALUATION

We conducted our experimental evaluation on a machine equipped with an Intel i7-6600U at 2.6 GHz providing the vector extension SSE with a vector register size of 128-bit. The capacities of the L1, L2, and L3-caches are 32KB, 256KB, and 4096KB, respectively. The system has 2 cores, however, we only investigate the single-thread performance. The size of the ECC DDR4 main memory is 19GB and all experiments happened entirely in-memory. We compiled our source code using `g++` with the optimization flag `-O3` on a 64-bit Linux operating system.

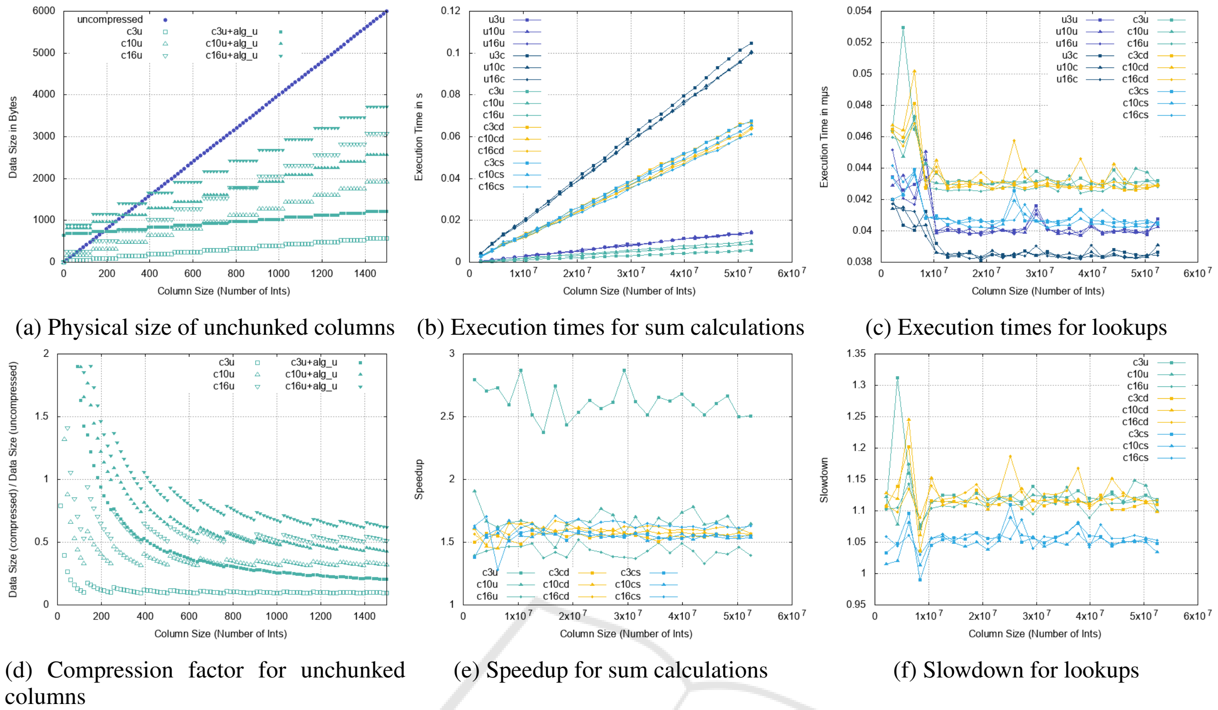
To show the benefit of our $Arrow^{Comp}$ compared to *Apache Arrow*, we evaluated (1) the data sizes of compressed and uncompressed arrays and (2) the runtimes of `sum` and `lookup` queries on uncompressed and compressed arrays. In our evaluation, we limit our focus on representative lightweight integer compression algorithm variants of SIMD-BP128 (cf. Figure 5). Moreover, we generated different columns with different bit widths per column, 3, 10, 16 (resp. different data ranges per column, $(0:2^3)$, $(0:2^{10})$, and $(0:2^{16})$). In detail, we compare five different column implementation types, marked by different colors:

- ■ uncompressed data, stored in one chunk resp. one single memory region,
- ■ uncompressed data, divided in chunks resp. memory regions containing 128 data values,
- ■ compressed data, binary packed with a given number of bits per data value in one single chunk,
- ■ compressed data, statically binary packed with a given number of bits per data value, chunk-wise, such that each chunk contains 128 values, and
- ■ compressed data, dynamically binary packed with a given number of bits per data value, chunk-wise, such that each chunk contains 8 bit containing the bit width used in this chunk, and 128 values.

We combined each of the three bit widths data sets with each of the five column types, such that for the experiments a table with 15 columns was used. In general, the unchunked uncompressed columns serve as the baseline for the unchunked compressed data and the chunked uncompressed columns serve baseline for the chunked (statically and dynamically) compressed columns. Statically compressed columns allow only one bit width per column. Dynamically compressed columns would principally allow different bit widths per chunk as proposed by SIMD-BP.

4.1 Data Sizes

The size of compressed data depends on (1) the number n of uncompressed integers, (2) the bit width b, and (3) the question, if the used bit width is stored additionally or not. If n is no multiple of 128, we use padding bits to complete the whole block. In this case, we do not store a bit width (■/■), the size (int Bytes) is calculated with $s_{-} = \frac{\lceil \frac{n}{128} \rceil \cdot 128 \cdot b}{8} + s_{meta}$. In the other case, due to storing a bit width per 128 values (■), the size is calculated by $s_{+} = \frac{\lceil \frac{n}{128} \rceil \cdot (128 \cdot b + 8)}{8} + s_{meta}$. Moreover, we have to include the size s_{meta} of algorithm descriptions which is in the of 640 to 680 KB for the used algorithms. The sizes for unchunked resp. statically compressed columns with and without the size of the algorithm descriptions are depicted in Figure 9(a) for column sizes 0 to 1500 values. The compression factor c (compressed data size/uncompressed data size) in dependence of the data size is shown in Figure 9(d). As we can see, with increasing column sizes, we can dramatically reduce the memory foot-



(a) Physical size of unchunked columns

(b) Execution times for sum calculations

(c) Execution times for lookups

(d) Compression factor for unchunked columns

(e) Speedup for sum calculations

(f) Slowdown for lookups

Figure 9: Evaluation of data sizes and execution times for `sum` aggregation and `lookup` queries.

print for all compressed variants depending on the bit width.

4.2 Run Times

To compare the runtimes of the `sum` aggregation (sequential access) and the lookup (random access), we varied the column sizes from $128 \cdot 2^{14}$ to $25 \cdot 128 \cdot 2^{14}$ logical integer values. Figure 9(b) shows the execution times for `sum` aggregation for uncompressed and compressed, unchunked and chunked columns. The execution time increases linearly in all cases. Execution times over unchunked columns (■/■) are faster than those on chunked columns (■/■/■). The comparison of the aggregation over unchunked uncompressed and compressed data (■) shows that, depending on the bit width, we can speed up the `sum` aggregation up to a factor of 2.5 (see Figure 9(e). The comparison of the aggregation over chunked uncompressed and compressed (■/■) columns shows that the processing yields a speedup of 1.5 for compressed values. And there is nearly no difference between statically and dynamically compressed columns, which means, that it is possible to hold differently compressed data in one column without a lack of performance.

For the lookup (random access), different positions in a column are given to extract the corresponding value. Similar to the `sum` aggregation experiment, column sizes are increased. The experiments were done with a warm cache, that is why the first measurements differ from the rest. Figure 9(c) shows, that the lookup does not depend on the column size. The fastest access can be measured by processing the uncompressed columns (■/■). Because, for compressed columns (■/■/■), the physical positions have to be calculated, which leads to an additional overhead. Lookups on dynamically compressed column (■) are the most expensive queries, because the bit width of each chunk has to be extracted before the values can be accessed. Lookups on statically compressed column (■) are less expensive than on the 1-chunk column, because the access of the right 128-value block is faster. An exception is bit width 16 in the unchunked column (■), because compiler optimizations lead to a better performance due to less bit-shifting operations and a minimal calculation amount to determine the physical position of the value. All in all, we are around 5% to 13% slower in lookups if we used compressed columns (see Figure 9(f)).

Nevertheless, OLAP queries usually do a lot of sequential access instead of random access. Thus, the integration of lightweight integer compression into *Apache Arrow* is very beneficial to reduce the memory footprint as well as to speed up the sequential access. Moreover, our self-describing compressed columns have a slightly higher memory consumption compared to only compressed columns, but this over-

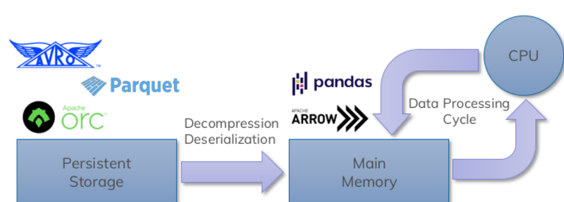


Figure 10: Data formats for disk and main memory storage.

head is negligible. As discussed in Section 3, random access can be supported in different ways. Nevertheless, more research in this direction is needed and compact data structures as presented in (Navarro, 2016) represent an interesting opportunity.

5 RELATED WORK

Related Work in this domain is manifold and our description is oriented according to the basic data flow process associated with an analytical application as shown in Figure 10. In this process, the data to be analyzed must be read from persistent storage and loaded into the main memory at first. Then, the second step is the processing step, where the actual analysis takes place. Both steps have different requirements and bottlenecks, so that different file formats exist for the persistent storage as well as the management in main memory as presented in Sections 5.1 and 5.2. As already described, *Apache Arrow* belongs to the file formats for the main memory. Additionally, there exists already some Apache Arrow-centric work as presented in Section 5.3.

5.1 Storage-oriented Formats

There are several storage-oriented file formats such as *Apache Parquet* (Vohra, 2016b), *Apache Avro* (Vohra, 2016a), or *Apache ORC* (Apache, 2020). These formats are widely used in the Hadoop ecosystem. While *Apache Avro* focuses on a row-based data format, *Apache Parquet* and *Apache ORC* are columnar formats like *Apache Arrow*. Because disk access is the bottleneck of conventional disk-based systems, the goal of those data storage architectures is to improve the disk access. This can be done by appropriate *heavyweight* data compression formats, which are characterized by there genericity and higher computational effort. For example, *Apache Parquet* supports snappy, gzip, lzo, brotli, lz4, and zstd to compress columnar data (Vohra, 2016b).

5.2 Processing-oriented Formats

Main memory or processing-oriented formats can be row-based for OLTP workloads, column-based for OLAP workloads, like *Apache Arrow* or *Pandas*, as well as hybrids. For example, *Pandas* is a python library for data analysis and manipulation. Here, heavyweight compression (gzip, bz2, zip, and xz) is used to reduce the amount of data for persistent storage in output formats like CSV, JSON, or TSV. For in-memory usage, no obvious compression is implemented. While reading the disk-oriented input format, the user can cast integral columns to datatypes like `int64`, `int32`, `int16`, or `int8` and initiate a kind of binary packing. Furthermore, columns can be load with the categorial datatype. Here, DICT is applied. And there exists a sparse series data type, which supports a null value suppression.

5.3 Apache Arrow-centric Work

The authors of (Peltenburg et al., 2019) created a Parquet-to-Arrow converter, which they implemented in FPGA. In their proposed scenario, the disk-oriented files are stored on fast accessible NVMe SSD's, which leads to the fact, that not the access to persistent storage but the conversion of file formats (persistent to in-memory) is a new bottleneck. The converter gets *Apache Parquet* files as pagewise input and constructs in-memory data structures in the *Apache Arrow* format using their Fletcher framework (Peltenburg et al., 2019).

With ArrowSAM (Ahmad et al., 2019), *Apache Arrow* is used to store SAM data (biological sequence-aligned to a reference sequence). The authors process genom data and apply several algorithms. To execute a parallel sorting algorithm, they split the genom data chromosome-wise in batches, which are added to the so-called *Apache Arrow* Plasma Object Store, a shared memory, that can be accessed by different clients without process boundaries.

6 CONCLUSION AND FUTURE WORK

Besides the development of a novel big data system, we also see a fundamental revision of the data organization and formats as a foundation for these systems. Here the goal is to organize data optimally for processing and analysis. In this context, *Apache Arrow* is a novel and highly interesting platform by providing a standardized language-independent colum-

nar in-memory format, organized for efficient analytical operations in modern hardware. It already includes dictionary coding to map arbitrary data types to integer values. However, *Apache Arrow* lacks support for more sophisticated lightweight integer compression algorithms which are suitable to (i) reduce the memory footprint and (ii) to speedup the columnar data processing. Thus, we presented an approach to integrate the large corpus of lightweight into *Apache Arrow* in this paper. We experimentally showed that this integration leads to a decreased memory footprint an increased performance of an aggregation function compared to uncompressed data.

The next step in our ongoing research work is the integration of more compression algorithms with different properties to generalize and optimize our integration approach. Another point of ongoing work is to deduce the decompression abstraction corresponding to compression metamodel. Thus, the generation of decompression code can be automated without the explicit knowledge of a decompression algorithm. Future work also includes the integration of some data hardening algorithms respectively error-detecting codes. This can be done by applying the metamodel as well. Last but not least, we plan to exhaustively evaluate the benefit of our approach with big data systems using *Apache Arrow*.

REFERENCES

Abadi, D., Boncz, P. A., Harizopoulos, S., Idreos, S., and Madden, S. (2013). The design and implementation of modern column-oriented database systems. *Foundations and Trends in Databases*, 5(3):197–280.

Abadi, D. J., Madden, S., and Ferreira, M. (2006). Integrating compression and execution in column-oriented database systems. In *SIGMOD*, pages 671–682.

Ahmad, T., Peltenburg, J., Ahmed, N., and Al Ars, Z. (2019). Arrowsam: In-memory genomics data processing through apache arrow framework. *bioRxiv*, page 741843.

Apache (2020). Apache avro: the smallest, fastest columnar storage for hadoop workloads. https://orc.apache.org/. Accessed: 2020-03-06.

Beazley, D. M. (2012). Data processing with pandas. *;login:*, 37(6).

Begoli, E., Camacho-Rodríguez, J., Hyde, J., Mior, M. J., and Lemire, D. (2018). Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *SIGMOD*, pages 221–230.

Binnig, C., Hildenbrand, S., and Färber, F. (2009). Dictionary-based order-preserving string compression for main memory column stores. In *SIGMOD*, page 283–296.

Boncz, P. A., Kersten, M. L., and Manegold, S. (2008). Breaking the memory wall in monetdb. *Commun. ACM*, 51(12):77–85.

Boncz, P. A., Zukowski, M., and Nes, N. (2005). Monetdb/x100: Hyper-pipelining query execution. In *CIDR*.

Chaudhuri, S., Dayal, U., and Narasayya, V. R. (2011). An overview of business intelligence technology. *Commun. ACM*, 54(8):88–98.

Damme, P., Habich, D., Hildebrandt, J., and Lehner, W. (2017). Lightweight data compression algorithms: An experimental survey (experiments and analyses). In *EDBT*, pages 72–83.

Damme, P., Ungethüm, A., Hildebrandt, J., Habich, D., and Lehner, W. (2019). From a comprehensive experimental survey to a cost-based selection strategy for lightweight integer compression algorithms. *ACM Trans. Database Syst.*, 44(3):9:1–9:46.

Damme, P., Ungethüm, A., Pietrzyk, J., Krause, A., Habich, D., and Lehner, W. (2020). Morphstore: Analytical query engine with a holistic compression-enabled processing model. *CoRR*, abs/2004.09350.

Goldstein, J., Ramakrishnan, R., and Shaft, U. (1998). Compressing relations and indexes. In *ICDE*, pages 370–379.

Google (2019). Snappy - a fast compressor/decompressor. https://github.com/google/snappy.

Habich, D., Damme, P., Ungethüm, A., Pietrzyk, J., Krause, A., Hildebrandt, J., and Lehner, W. (2019). Morphstore - in-memory query processing based on morphing compressed intermediates LIVE. In *SIGMOD*, pages 1917–1920.

Hildebrandt, J., Habich, D., Damme, P., and Lehner, W. (2016). Compression-aware in-memory query processing: Vision, system design and beyond. In *ADMS@VLDB*, pages 40–56.

Hildebrandt, J., Habich, D., Kühn, T., Damme, P., and Lehner, W. (2017). Metamodeling lightweight data compression algorithms and its application scenarios. In *ER*, pages 128–141.

Huffman, D. A. (1952). A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101.

Kornacker, M., Behm, A., Bittorf, V., Bobrovytsky, T., Ching, C., Choi, A., Erickson, J., Grund, M., Hecht, D., Jacobs, M., Joshi, I., Kuff, L., Kumar, D., Leblang, A., Li, N., Pandis, I., Robinson, H., Rorke, D., Rus, S., Russell, J., Tsirogiannis, D., Wanderman-Milne, S., and Yoder, M. (2015). Impala: A modern, open-source SQL engine for hadoop. In *CIDR*.

Lemire, D. and Boytsov, L. (2015). Decoding billions of integers per second through vectorization. *Softw., Pract. Exper.*, 45(1):1–29.

Müller, I., Ratsch, C., and Färber, F. (2014). Adaptive string dictionary compression in in-memory column-store database systems. In *EDBT*, pages 283–294.

Navarro, G. (2016). *Compact Data Structures - A Practical Approach*. Cambridge University Press.

Peltenburg, J., van Straten, J., Wijtemans, L., van Leeuwen, L., Al-Ars, Z., and Hofstee, P. (2019). Fletcher: A

framework to efficiently integrate FPGA accelerators with apache arrow. In *FPL*, pages 270–277.

Polychroniou, O., Raghavan, A., and Ross, K. A. (2015). Rethinking SIMD vectorization for in-memory databases. In *SIGMOD*, pages 1493–1508.

Polychroniou, O. and Ross, K. A. (2019). Towards practical vectorized analytical query engines. In *DaMoN@SIGMOD*, pages 10:1–10:7.

Roth, M. A. and Van Horn, S. J. (1993). Database compression. *SIGMOD Rec.*, 22(3):31–39.

Sridhar, K. T. (2017). Modern column stores for big data processing. In Reddy, P. K., Sureka, A., Chakravarthy, S., and Bhalla, S., editors, *BDA*, pages 113–125.

Stonebraker, M., Abadi, D. J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E. J., O'Neil, P. E., Rasin, A., Tran, N., and Zdonik, S. B. (2005). C-store: A column-oriented DBMS. In *VLDB*, pages 553–564.

Ungethüm, A., Pietrzyk, J., Damme, P., Krause, A., Habich, D., Lehner, W., and Focht, E. (2020). Hardware-oblivious SIMD parallelism for in-memory column-stores. In *CIDR*.

Vohra, D. (2016a). Apache avro. In *Practical Hadoop Ecosystem*, pages 303–323. Springer.

Vohra, D. (2016b). Apache parquet. In *Practical Hadoop Ecosystem*, pages 325–335. Springer.

Witten, I. H., Neal, R. M., and Cleary, J. G. (1987). Arithmetic coding for data compression. *Commun. ACM*, 30(6):520–540.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010). Spark: Cluster computing with working sets. In Nahum, E. M. and Xu, D., editors, *HotCloud*.

Zhou, J. and Ross, K. A. (2002). Implementing database operations using SIMD instructions. In *SIGMOD*, pages 145–156.

Ziv, J. and Lempel, A. (1977). A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theor.*, 23(3):337–343.

Ziv, J. and Lempel, A. (1978). Compression of individual sequences via variable-rate coding. *IEEE Trans. Information Theory*, 24(5):530–536.

Zukowski, M., Héman, S., Nes, N., and Boncz, P. A. (2006). Super-scalar RAM-CPU cache compression. In *ICDE*, page 59.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS