Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Towards a New File Format for Big Data: SIMD-Friendly Composable Compression

Author: Azim Afroozeh (2639408)

1st supervisor: Prof. Peter Boncz
2nd reader: Dr. Marc Makkes

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

October 30, 2020

I dedicate this thesis to my family, for their encouragement and support at every step during my Master's studies.

Abstract

In this thesis, we review the state-of-the-art big data file formats such as Parquet and ORC and justify the need for designing a new file format. We argue that light-weight compression schemes such as RLE, PFOR, PDelta and PDICT are better candidates than general-purpose compression schemes to be used in a big data file format. However, these compression schemes are not designed to leverage SIMD instructions. Considering the fact that there are SIMD instructions available that can operate on 64 integers in one cycle, and the availability of wider SIMD registers in the future, it is necessary to redesign these compression schemes to be SIMD-friendly. Moreover, these compression schemes are not flexible to provide the choice of outlier handling mechanism and be recursively combined.

To solve these problems, we propose the SIMD-friendly composable compression model, which can be considered as the foundation of a new file format for big data. To be SIMD-friendly, we propose several new layouts that make these compression schemes capable of exploiting the widest SIMD register supported by a CPU (even future CPUs). Moreover, the decompression function is decomposed into several efficient functions, which can be combined later to provide the desired flexibility, and the ability to be recursively combined. The results show that the overhead of having light-weight compression schemes such as PFOR, PDelta, PDICT, and RLE in the composable model is negligible, and SIMD instructions can accelerate the decompression phase to 50 tuples per cycle. This is a performance improvement of two orders of magnitude compared to the predecessor techniques.

Acknowledgements

This thesis would not have been possible without the guidance and support of my supervisor, Professor Peter Boncz. I would like to thank him for the invaluable discussions and advice in the past 9 months, which has shaped the work in this thesis. I particularly enjoyed his vast knowledge of database systems and performance optimization. I am looking forward to start working as a Ph.D. student with him. Furthermore, I would like to thank Dr. Marc Makkes and Dr. Mark Raasveldt, for their useful feedback. This thesis was carried out as an internship at Centrum Wiskunde & Informatica (CWI), which provided an excellent research environment. Unfortunately, due to the global pandemic circumstances, I had to work from home most of my internship. I am thankful to the CWI personnel for their assistance and support during this period and looking forward to working at CWI next year.

Contents

List of Figures **v**

List of Tables **ix**

1 Introduction **1**

1.1 Big Data File Formats 1
1.2 Shortcomings of the State-of-the-art Big Data File Formats 2
1.3 Background 4
1.4 A New File Format 9
1.5 Research Questions 11
1.6 Outline 12

2 Related Work **13**

2.1 Storage Layouts 13
2.1.1 NSM 13
2.1.2 DSM 13
2.1.3 PAX 14
2.2 Bit-(Un)Packing 15
2.2.1 Vectorized Bit-Unpacking 16
2.2.2 BitWeaving 17
2.3 Light-Weight Compression Algorithms 19
2.4 Frame of Reference 20
2.4.1 PFOR 20
2.4.2 AFOR 22
2.4.3 VSEncoding 23
2.4.4 NewPFD and OptPFD 23
2.4.5 ParaPFOR 24
2.4.6 SIMD-FASTPFOR, FASTPFOR and SIMPLEPFOR 24

2.4.7 S4-FASTPFOR . . . 25
2.5 Delta Coding (Differential Coding) . . . 25
2.5.1 D4, DM, D2, D1 . . . 26
2.5.2 Vectorized Prefix Sum . . . 26
2.5.3 PDelta and PFOR-Delta . . . 27
2.5.4 V-PFORDelta . . . 27
2.6 Dictionary Encodings . . . 27
2.6.1 PDICT . . . 28
2.7 RLE . . . 28
2.7.1 Vectorized RLE . . . 28
2.8 Fast Static Symbol Table . . . 29
2.9 General SIMD-based Compression algorithms . . . 29
2.10 Whitebox Compression . . . 30
2.11 Big Data File Formats . . . 30
2.11.1 Parquet . . . 31
2.11.2 ORC . . . 33
2.11.3 Artus . . . 35
2.11.4 Data Blocks . . . 35
2.11.5 Albis . . . 36
2.12 Matrix Tranpose . . . 37

3 SIMD-Friendly Bit-(Un)Packing 39

3.1 Storage Layout . . . 40
3.1.1 Intermediate Word . . . 41
3.2 Bit-unpacking . . . 41
3.2.1 Scalar Bit-Unpacking . . . 42
3.2.2 Auto Vectorization . . . 45
3.2.3 Vectorized Bit-Unpacking . . . 45
3.3 Evaluation . . . 48
3.3.1 Scalar Unpacking . . . 48
3.3.2 Vectorized Unpacking and Intermediate Word . . . 50
3.3.3 Auto Vectorization . . . 51

4 Composable Functions **55**
4.1 Bit-Unpacking . . . 55
4.1.1 Evaluation . . . 55
4.2 Prefix Sum . . . 56
4.2.1 Transposed Layout . . . 57
4.2.2 Evaluation . . . 59
4.3 Plus . . . 66
4.3.1 Intermediate Base . . . 66
4.3.2 Evaluation . . . 66
4.4 Transpose . . . 67
4.4.1 Evaluation . . . 72
4.5 Load . . . 73
4.6 Shift . . . 73
4.6.1 Evaluation . . . 74
4.7 Patch . . . 75
4.7.1 Evaluation . . . 76
4.8 Split . . . 76
4.8.1 Patch substitution . . . 77
4.8.2 Null support . . . 77
4.8.3 Implementation and Evaluation . . . 78
4.9 RLE . . . 85
4.9.1 SIMD RLE Layout . . . 86
4.9.2 Intermediate Representation . . . 87
4.9.3 Decompression Algorithm . . . 91
4.9.4 Evaluation . . . 94

5 Composable Compression Schemes **97**
5.1 Composable PFOR . . . 97
5.1.1 Evaluation . . . 98
5.2 Composable PDelta . . . 99
5.3 Composable PFOR-Delta . . . 100
5.4 Composable PDICT . . . 100
5.4.1 Evaluation . . . 102

6 Conclusions **105**

6.1 Research questions 106

6.2 Future Work 109

References **111**

Appendices **119**

A.1 C-PFOR 119

A.2 Literature Study 123

List of Figures

1.1 An example of the whitebox page layout. 5
1.2 An example of the whitebox page layout where an integer column is compressed by the enhanced FOR. 6
1.3 An example of the whitebox page layout where 4096 integers are compressed by the enhanced FOR. 7
1.4 An example of the whitebox file format where 8 pages with same layout are stored consecutively. 8
1.5 An example of the whitebox page layout where data size is small. 8
1.6 An example of the whitebox page layout in the PAX format. 9
1.7 The components of a new big data file format. Note that The scope of this thesis is the base layer, namely the SIMD-friendly compression and the composable compression and the other components will be investigated in future work. 10

2.1 An example of an NSM page (this figure is borrowed from the paper by Ailamaki *et al.* (1)) 14
2.2 An example of an Pax page (This Figure is borrowed from the paper by Ailamaki *et al.* (1)) 15
2.3 An example of Bit-Unpacking 16
2.4 4-way Vertical layout (This figure is borrowed from the paper by Schlegel *et al.* (2)) 17
2.5 An example of Whitebox compression (This Figure is borrowed from the paper by Ghita *et al.* (3)) 31
2.6 The Parquet file format 32
2.7 The ORC file format (This Figure is borrowed from the paper by Huai *et al.* (4)) 34

2.8 Layout of a Data Block for n attributes (This figure is borrowed from the paper by Lang *et al.* (5)) 36
2.9 An example of a Row format in Albis 37
2.10 An Example of table partitioning in Albis 37
2.11 An example of the matrix transpose 38

3.1 The 1024-bit interleaved data layout where W = 32 and B = 7. 41
3.2 The 1024-bit interleaved data layout where W = 32 and B = 7. 41
3.3 An example of the multi cursors technique applied to scalar bit-(un)packing. 42
3.4 Scalar bit-unpacking performance measured in terms of cycles per tuple for each B and W 49
3.5 The performance of the non-interleaved layout vs interleaved layout in terms of cycles per tuple. 49
3.6 The performance of vectorized bit-unpacking in terms of cycles per tuple for each R (number of cursors). 50
3.7 The performance of vectorized bit-unpacking in terms of cycles per tuple for each B and W. 51
3.8 The performance of vectorized bit-unpacking in terms of cycles per tuple for each B and W. 52
3.9 The performance of auto-vectorized bit-unpacking in terms of cycles per tuple for each possible B, W and R. 53
3.10 The performance of auto-vectorized bit-unpacking vs explicitly vectorized bit-unpacking in terms of cycles per tuple for each possible B, W. 54
3.11 The performance of auto-vectorized bit-unpacking vs explicitly vectorized bit-unpacking in terms of cycles per tuple on M1, where $W = 64$; 54

4.1 Difference between scenario A and B in terms of cycles per tuple. 56
4.2 An example of a prefix-sum function. 57
4.3 The transposed layout for the prefix-sum calculation. 58
4.4 The 2-cursors layout for prefix-sum calculation 58
4.5 The performance of different implementation of prefix-sum in terms of cycles per tuple. 62
4.6 The performance of the plus function combined with bit-unpacking vs the separate version in terms of cycles per tuple. 67
4.7 Our new transpose algorithm, the loading step. 69
4.8 Our new transpose algorithm where $i = 0$. 70

4.9 Our new transpose algorithm where $i = 1$. 70
4.10 Our new transpose algorithm where $i = 3$. 71
4.11 Our new transpose algorithm where $i = 7$. 71
4.12 Our new transpose algorithm, final result. 72
4.13 The result of different implementations of the transpose algorithm. 73
4.14 The performance of the bit-unpack function combined with shift and plus vs the bit-unpack function combined with the plus vs the bit-unpack function in terms of cycles per tuple. 74
4.15 An example of the patch function. 75
4.16 The result of the patch function. 76
4.17 An example of all variations of the split function. 77
4.18 The result of different implementations of the split function. 79
4.19 The result of different implementations of the split function for the patch case. 82
4.20 The result of different implementations of the split function for the Null case. 85
4.21 An example of our RLE layout where the length and value columns are stored separately. 86
4.22 An example of SIMDizing our new RLE layout. 87
4.23 The layout of intermediate RLE. 88
4.24 An example of the intermediate RLE layout. 89
4.25 The result of different layouts for RLE in terms of number of bits used to compress data for different number of runs. 95
4.26 The result of different layouts for RLE in terms of compression ratio for different number of runs. 95
4.27 The result of different layouts for RLE in terms of compression ratio for different number of runs. 96
4.28 The performance of different layouts for RLE in terms of number of cycle per tuple for different number of runs. 96

5.1 The function composition for C-PFOR. 98
5.2 Difference between scenario A and B in terms of compression ratio. 99
5.3 The function composition for composable PDelta. 100
5.4 The function composition for composable PFOR-Delta. 100
5.5 The function composition of C-PDICT, variation 1. 101
5.6 The function composition of C-PDICT, variation 2. 102

5.7 The performance of C-PFOR vs PFOR in terms of cycles per tuple for each possible B and W. The exception ratio is 0. 103
5.8 The performance of C-PFOR vs PFOR in terms of cycles per tuple for each possible B and W. The exception ratio is 0.05. 104

List of Tables

2.1 Overview of all variants of FOR 21

3.1 Specifications of the machines used for benchmarking. 48

1 Difference between scenario A and B in terms of compression ratio. 122
2 Literature inclusion table. 126
3 Inclusion and exclusion criteria 127

1

Introduction

1.1 Big Data File Formats

Several relational data processing systems such as Apache Spark (6), Apache Hive (7), and Apache Impala (8) have been emerged to analyze so-called big data. These systems manage their data in a variety of open data file formats such as Parquet (9) and ORC (10). Parquet is the most popular file format to store data in the Cloud (11).

Over the last decade, several factors of the big data landscape have changed, which might affect the suitability of the state-of-the-art big data file formats. First, as software ecosystem becomes more mature, better metadata is required. In database systems, metadata such as zone maps, table samples and Hyperloglog (12) is required for cardinality estimation (a critical piece in query optimization) as well as to enable targeted data access (skipping whole files or part of a file). Second, continuous progress in hardware, such as wider Single Instruction Multiple Data (SIMD) registers and more advanced instructions offer new opportunities for fast data access (13). Third, it has been realized that data or the structure of data (schema) continuously changes (14). Therefore, the ability to efficiently handle all types of changes, such as schema evolution and updates has increasingly become a requirement for big data file formats. Lastly, the shift from the Hadoop ecosystem to the Cloud introduces new challenges for big data file formats from both technical and privacy perspectives (15). The mentioned trends raise the question if there is a need to design a new big data file format, which is the topic of this thesis.

1.2 Shortcomings of the State-of-the-art Big Data File Formats

Traditionally, data has been stored in databases row by row, which is called the N-ary Storage Model (NSM). While being a good fit for random-access algorithms, the NSM model's performance degrades for sequential scenarios (16). A model proposed to solve this issue is the Decomposition Storage Model (DSM) (17), in which data is stored column by column. DSM provides column pruning opportunities to skip reading unwanted columns and it also favors compression since similar data are put together (18). Another alternative is the Partition Attributes Across (PAX) model (19), which is the combination of the two mentioned models. PAX stores chunks of rows in a columnar fashion.

Parquet is a compressed, columnar data storage that is based on the PAX data model. A Parquet file consists of one or more *row-groups*. A row-group is a logical horizontal partitioning of the data into rows that precisely contains one *column chunk* per column. A column chunk contains a chunk of the data for a particular column. Column chunks are divided into pages that are conceptually an indivisible unit in terms of compression and encoding. Note that the metadata in a Parquet file is stored as a footer.

While the state-of-the-art big data file formats such as Parquet provide a relatively good data compression ratio and efficient scan, they suffer from several shortcomings that we elaborate on in this section.

Efficient Implementation The design of Parquet has been influenced by the Java Virtual Machine (JVM), which in contrast to lower level programming languages such as C and C++, does not allow precise control of system behavior, such as memory allocation, and also does not provide an intrinsics interface that exposes SIMD instructions (20). Moreover, Parquet V1, the first version of Parquet, in terms of encoding scheme only offers dictionary encoding (9), where the codes are subsequently stored in a compact format using either run-length encoding or bit packing. The combination of these particular techniques may work well in some data distribution, but can be a mismatch or overkill in other situations. It is worth mentioning that Parquet V2, the second version of Parquet, adds more diverse encoding schemes. However, we should note that by far most data in the Cloud use Parquet V1.

In addition, dynamically switching between bit-packing and run-length encoding in Parquet can cause branchy decoding that prevents compiler optimizations and may lead to branch miss-prediction penalties. Lastly, bit-packing used in Parquet could be improved

to be more SIMD-friendly using interleaving. Based on these reasons, there is a need to design a new file format that enables us to exploit SIMD opportunities whenever possible, to be able to provide more encoding schemes with better implementations which can select the most appropriate scheme based on the characteristics of the data.

Page Compression Parquet and ORC use general-purpose compression techniques to compress pages. We argue that domain-specific compression methods such as PFOR, PDELTA, and PDICT are a better choice for non-string data (21). In addition, the new string compression scheme FSST (22) allows predicate pushdown on compressed string data, while offering competitive compression ratios and (de)compression speed. Therefore, domain-specific and string compression techniques such as FSST can have a significant impact on compression ratios and (de)compression speed. This justifies the need for designing a new file format that uses such compression techniques.

Wide and Sparse Data Parquet treats wide and sparse data similar to other kinds of data, which leads to several problems. In Parquet, the page size is fixed causing the page to be nearly empty in case of wide and sparse data, which is not efficient from both the space and computation speed perspectives. Moreover, if at least one page needs to be reserved during the append operation, a fixed-sized page implies a large RAM buffering requirement while calling the writer API.

Design Some of the design decisions made by Parquet could be improved. First, a field containing the size of the row-group header could be added to the row-group header. Currently, the Parquet metadata parser needs to guess the right size of the row group header, which might take several tries to guess correctly (23). Second, the number of values in a page could be added as metadata. This improvement enables us to skip pages in cases where specific rows needs to be fetched by row id. In addition, in Parquet, row-group statistics are embedded in the row-group header, which is problematic because of the row-group header size, as already discussed.

Whitebox Compression The new proprietary Artus (24) data format by Google directly exposes dictionary-compressed columns as an integer column of codes and a dictionary object. This allows consumers to delay decompression which results in performance gains in multiple cases such as predicate pushdown. Besides dictionary-compressed columns, Artus also exposes RLE compressed data as a two-column stream of count and

value pairs which results in a better performance. Taking this further in the recent work on white-box compression (25) by CWI's Database group, we argue that the new file format for big data should store a table as a number of optimized physical columns. In addition, a mapping of physical to logical columns, which the user expects in the table, should be provided in the header. In fact, white-box compression can be applied in a broader context than the two discussed features of Artus, which can allow for better compression ratios and faster processing.

Cloud Storage The metadata in Parquet and ORC is located in row-group headers or global footers. In Cloud storage, in contrast to HDFS (26), this leads to buckets where data is split across many files, one per append. Also, random access into many files for small pieces of metadata is not favorable in the Cloud. We argue that a new design is needed to overcome this problem by separating the metadata from the actual data. All metadata should be put together in one file to achieve a more efficient access pattern, particularly in the workloads with pruning possibilities.

1.3 Background

At CWI's Database Architectures group, we have started to design a new big data format. As the first step, we designed a new page layout to support the white-box compression model. The white-box compression model represents logical columns, what the user expects to see, as composite *functions* of physical columns, what is stored on the disk. These functions are standard column expressions that are fast and expressive enough to handle different patterns (3).

To store a table in this page layout, the table is horizontally partitioned into multiple *extents*. An extent is similar to a row-group in Parquet as it contains all the attributes of a row. Each extent consists of multiple consecutive pages, similar to a page in Parquet. We propose the default fixed size of 256 KB for a page. During bulk loading, 8 consecutive pages are filled per column. This results in having 2 MB of consecutive data that favors big scans. Figure 1.1 shows an example of our new page layout.

In this page layout, each page consists of one or more *frames*. A frame is the base representation for any tabular data and stores a table slice with all its data that reside inside one page. In case where a frame holds multiple columns, it resembles the PAX layout. Moreover, each page consists of one *page footer* located at the end of the page.

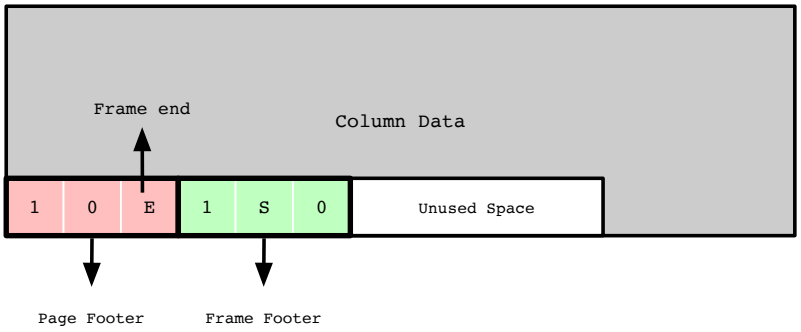


Figure 1.1: An example of the whitebox page layout.

The page footer denotes the number of frames and their end-offset in a page, all encoded as a 4-byte integer. Note that all these offsets are relative to the start of the page.

A frame has one or more *segments*, which is the primary storage unit that stores data of a single physical column. Besides segments, each frame has one *frame footer* located at the end of the frame. A frame footer indicates the number of segments, their size, and their start offset in a frame, all encoded as a 4-byte integer. It is worth mentioning that all these offsets are relative to the beginning of the current frame. Lastly, a special form of a frame, called mini-frame, is defined to store in-page metadata which contains one row per 1024 rows in the current frame. The mini-frame is appended to the frame and can be considered as a mini table that makes our page format flexible enough to store any kind of metadata for a column. For example, a mini-frame allows us to store the bit-width and the base value for each 1024-row vector necessary to optimize PFOR.

A segment comes in two flavors: compressed and uncompressed. A compressed segment stores bit-packed data using the 1024-bits interleaved layout (see Section 3). In our 1024-bits interleaved layout, 1024 B-bit tuples are interleaved in L lanes in B 1024-bits wide words. Each lane is $W = 1024/L$ bits wide, where W is 8, 16, 32 or 64. A tuple is bit-packed and appended to the lane $X \ mod \ L$. An uncompressed segment stores fixed-sized physical type values in an array. An example of an uncompressed segment is an entry-frame, which resembles a page-entry. Note that a frame might consist of a mix of compressed and uncompressed segments.

Variable-sized data cannot be stored in one segment. Instead, it should be stored in two segments: one segment with an unsigned integer byte-offset, and another segment with the actual data represented in bytes. Moreover, if a segment contains FSST compressed strings, a 2KB FSST lookup table needs to be stored in a separate segment.

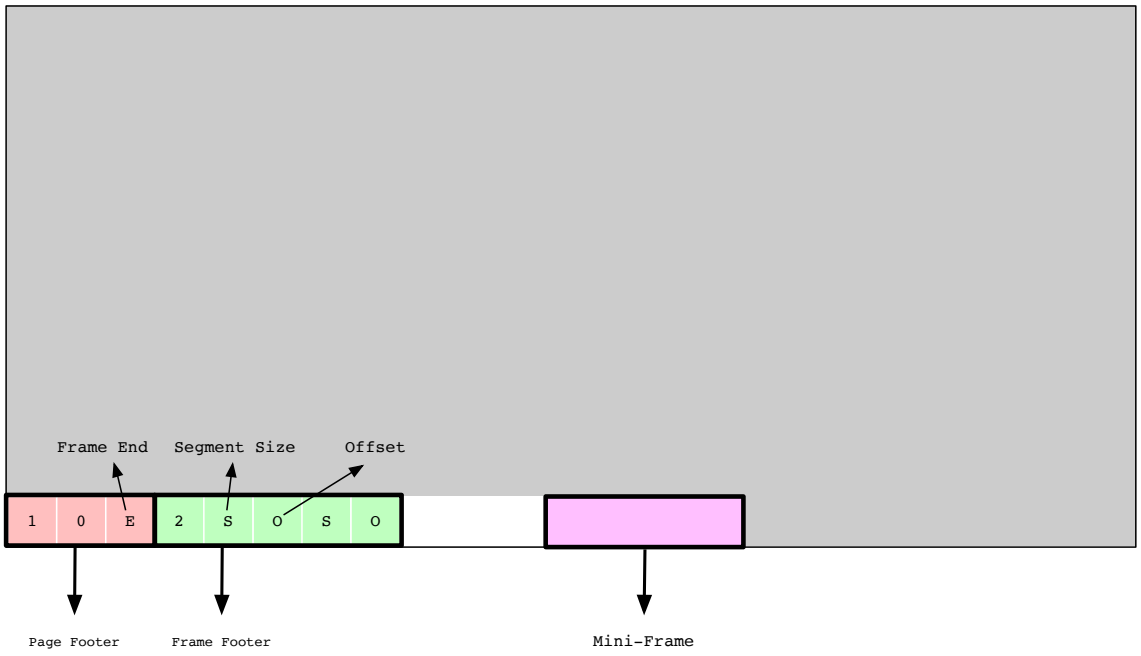


Figure 1.2: An example of the whitebox page layout where an integer column is compressed by the enhanced FOR.

To better illustrate how our page layout works, consider an integer column that is compressed using the Frame of Reference (FOR) compression scheme. In FOR, a single base value is determined for the whole page and is subtracted from integers in order to store them with fewer bits. The FOR scheme in this example is enhanced by having different base values for every 1024 tuples, which increases the flexibility of FOR. This page layout enables us to have this enhancement by storing the base values in the mini-frame. Figure 1.2 shows an integer column that is stored in our new page layout. In Figure 1.2, the green boxes show the content of the frame footer. A box with the letter S denotes the size of a segment, while the box with the letter O denotes the offset pointing to the beginning of a segment. This frame originally consists of a single compressed segment denoted by the color gray. As data needs in-page metadata, a mini-frame containing metadata is created and appended to the frame. The mini-frame consists of an uncompressed segment which stores bases for every 1024 tuples. Therefore the number of segments inside the frame is increased to 2 and denoted by the first green box in the frame footer. The red boxes show the content of the page footer, in which the first box shows the number of frames inside this page. The last box inside the page footer points to the end of the frame, where the

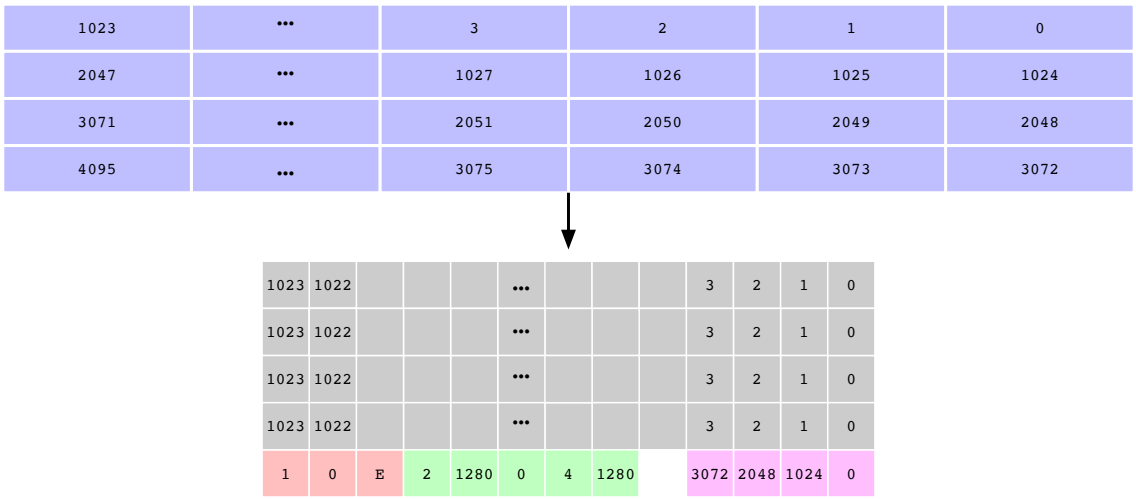


Figure 1.3: An example of the whitebox page layout where 4096 integers are compressed by the enhanced FOR.

API reader could find the number of segments. Note that each box represents a 32-bit integer.

To exactly show how the previous example works, consider a column filled with 4096 32-bit integers with values 0 to 4095, as shown in Figure 1.3. To compress the integers in this column via enhanced FOR and store them in our new page layout, the following step needs to be taken. First, for every 1024 tuples, a base value needs to be determined, which is the tuple with the smallest value in 1024 tuples. Therefore, base values for our example are 0, 1024, 2048, and 3072, respectively. Second, the integers need to bit-packed after the base value is subtracted from them. Note that the base is different for every 1024 tuples. As we store only one column in the page layout, only one frame is needed. Third, the compressed data needs to be stored in a segment, and subsequently, the offset and size of the segment need to be added to the frame footer. The offset of the current segment is 0, while the size of the segment is 1280 as it stores 1280 32-bit integers (only 10 bits is needed to represent each tuple, therefore $4096 * 10$ bits or 40960/32 integers are needed). Finally, the base values, which can be considered as in-page metadata, need to be stored inside a segment stored inside a mini-frame. Then, this mini frame needs to be appended to the first frame, which contains the actual data, and the offset and size of the segment inside the mini-frame need to be added to the frame footer. The offset of this segment is 1280, with the size of 4 as it stores 4 32-bit integers.

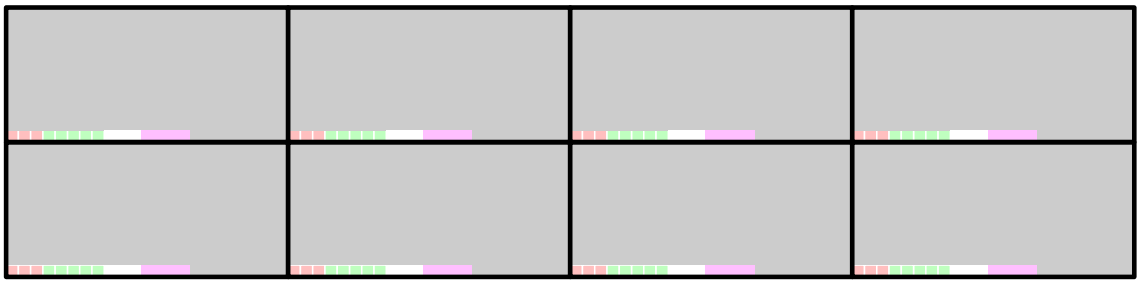


Figure 1.4: An example of the whitebox file format where 8 pages with same layout are stored consecutively.

To achieve better sequential bandwidth, both on magnetic drives as well as in cloud storage, 8 consecutive pages are filled with this layout, as illustrated in Figure 1.4. In cases when the integer column in our example contains only a few tuples, the data from other columns need to be filled in the page layout to avoid increasing the unused space inside a page. An example of this case is illustrated in Figure 1.5. As can be seen, there are 3 frames and 4 segments inside the page.

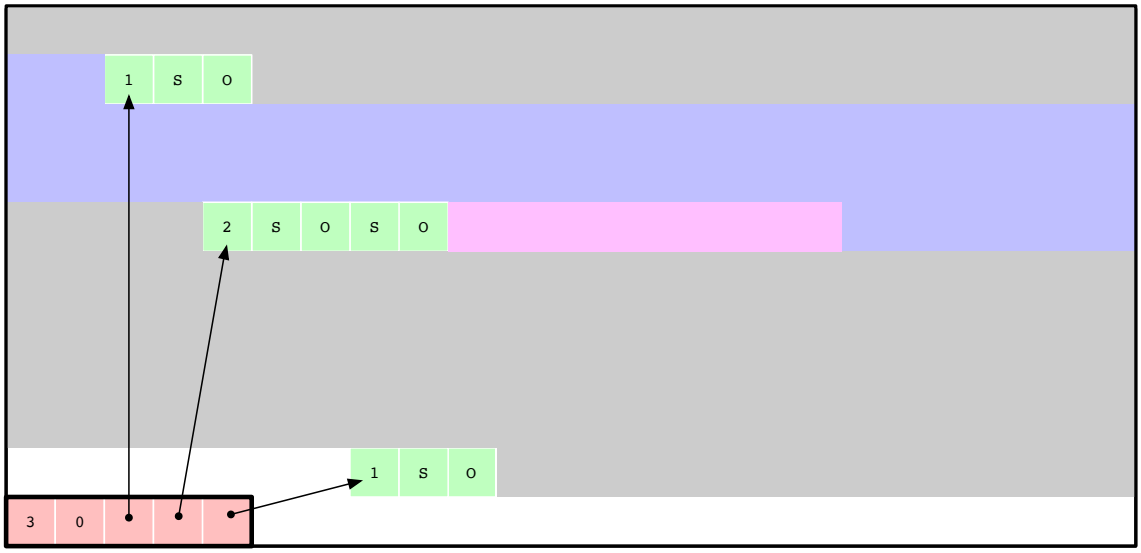


Figure 1.5: An example of the whitebox page layout where data size is small.

Finally, in Figure 1.6, the layout of the previous example is illustrated in the PAX format. As can be seen, all frames are combined into one frame, which has a piece of data

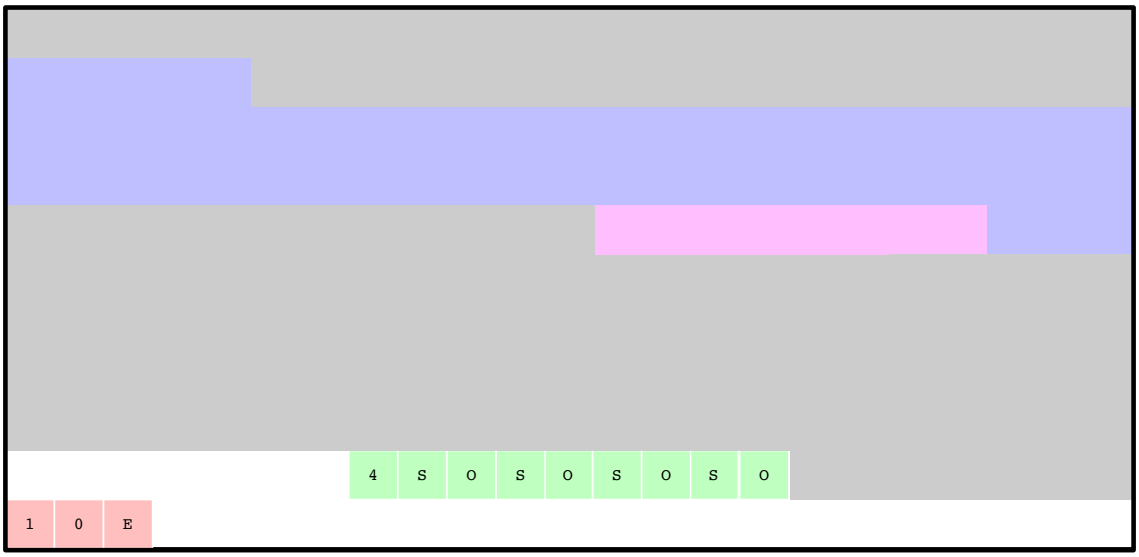


Figure 1.6: An example of the whitebox page layout in the PAX format.

from all columns.

1.4 A New File Format

To design a new big data file format that overcomes the shortcomings of the state-of-the-art big data file formats and is compatible with new emerging trends in the software ecosystem, many different components need to be designed and implemented. Figure 1.7 shows various component that are required to design a new file format. The bottom layer, the SIMD-friendly compression and combosable compression form the foundation of a new file format on which the other components can be built. The scope of this thesis is the base layer, namely the SIMD-friendly compression and the composable compression. The other components will be investigated in future work.

We argue that light-weight compression schemes such as RLE, PFOR, PDelta and PDICT are good candidates to be used in a big data file format. However, these compression schemes are not designed to leverage SIMD instructions. Currently, there are SIMD intrinsics available that can operate on 64 32-bit registers in a single instruction (27). This leads to 64 times more parallelism and performance improvement. Considering that even wider SIMD registers will be released in the future, it is necessary to redesign these compression schemes to be SIMD-friendly. We call a compression scheme SIMD-friendly,

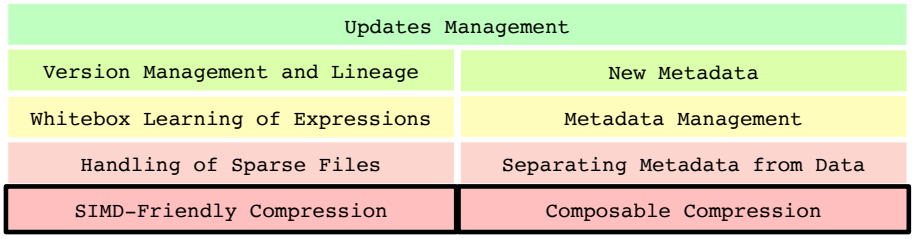


Figure 1.7: The components of a new big data file format. Note that The scope of this thesis is the base layer, namely the SIMD-friendly compression and the composable compression and the other components will be investigated in future work.

if during the decompression, it can exploits the instructions for the widest register that a CPU supports, both for the currently available CPUs and CPUs with wider SIMD registers in the future.

For example, bit-unpacking is the main component of the mentioned light-weight compression schemes. However, the lack of SIMD-friendly bit-unpacking methods forced database designers to completely avoid bit-packing in their data formats (5). In this thesis, we design a completely SIMD-friendly bit-unpacking and, on top of that, SIMDize other parts of light-weight compression schemes.

To implement SIMD-friendly compression schemes, explicit SIMD intrinsics can be used. However, the usage of explicit SIMD intrinsics makes the process of maintaining the code harder. One possible solution is to implement the SIMD-friendly compression without any explicit SIMD instructions and be dependent on the compiler to automatically detect and replace parts of the code with SIMD instructions, when there is an opportunity. However, whether the compilers are able to completely detect these opportunities is research question that is investigated in this thesis.

Most of the light-weight compression schemes use the patching technique to handle outliers, which in some cases, e.g., for data that can be represented by using few bits, has limitations and needs to be replaced by another outlier handling mechanism. This can be solved by having two different implementations for every compression scheme, one with the patching technique and one with an alternative. However, this solution increases the complexity of the system. Moreover, in some cases, e.g., where string data is represented by an array of bytes and offsets, we know for sure that there is no outlier in offsets, even if the data changes. In such cases, the patching phase in the patched leads to both computation and space overhead and needs to be removed.

Furthermore, to achieve the best possible compression ratio, in some cases, compression schemes need to be recursively combined. For example, in case of patched encodings, the exceptions are stored without being compressed, while they can be compressed. Besides exceptions, in case of RLE, if the value and the length of each run are separated, both of them are a good candidate to be further compressed.

Having a flexible choice of patching, and supporting recursive compression are two reasons that the light-weight compression schemes, which are usually implemented as one function which does multiple different tasks at the same time, need to be decomposed into several composable components.

In the whitebox compression model, the decompression is decomposed to simple, transparent functions that perform only one task (3). Now the question arises if it is possible to apply the whitebox compression principles to blackbox compression schemes to achieve the desired flexibility without performance penalty? In other words, to what extent we should whitebox or decompose the compression functions to achieve the desired flexibility without a performance penalty. This question is investigated in this thesis.

1.5 Research Questions

The goal of this thesis is to investigate the following research questions, which are categorized into the following four broad categories:

1. How can compression functions be decomposed to multiple functions to be flexible and efficient at the same time? Is it possible to apply the whitebox compression principles to blackbox compression schemes to achieve the desired flexibility without performance penalty?

2. The patching technique, which was proposed to mitigate the light-weight compression vulnerability to outliers, leads to a worse compression ratio in cases where only a few bits are required to represent data. What is a good alternative to the patching technique to handle outliers in such cases?

3. How can we make compression schemes such as PFOR, PDelta, PDICT and RLE SIMD-friendly so that the compression scheme can exploit the instructions for the widest register that a CPU supports, both for the currently available CPUs and CPUs with wider SIMD registers in the future?

4. Does our new SIMD-friendly compression schemes need to be implemented using explicit SIMD intrinsics? Is auto vectorization offered by compilers, such as GCC, able to obtain the performance of our explicit implementation?

1.6 Outline

The rest of this thesis is organized as follows. In Chapter 2 we give an overview of the related work, with focus on compression algorithms in file formats. In Chapter 3 we discuss our SIMD-friendly bit-(un)packing technique. To support the light-weight comression schemes in the whitebox compression model, several functions need to be added. In Chapter 4, we describe these function and investigate if these functions are capable of delivering the same performance as they were combined with each other in the blackbox compression model. In Chapter 5, we show how light-weight compression schemes could be composed using these functions to to be flexible and efficient at the same time. Note that instead of a separate evaluation chapter, we discuss the experiment and the result of the experiment for each section in the section itself. Finally, in Chapter 6, we conclude this thesis by revisiting the research questions and answering them.

2

Related Work

In this section, we discuss the related work necessary for understanding the contributions of this thesis. This section is organized as follows. In Section 2.1, we discuss database storage layouts. Bit-packing, the building block of most light-weight compression algorithms, is discussed in Section 2.2. Then, we give an overview of light-weight compression algorithms, which are popular in column-oriented databases, the frame of reference, delta coding, dictionary coding, run-length encoding, and fast static symbol table. We discuss a general approach to SIMDize all compression algorithms in Section 2.9, and conclude this section by reviewing white-box compression, big data file formats and matrix transpose.

2.1 Storage Layouts

2.1.1 NSM

Traditionally, in database systems, tabular data has been stored in database systems row by row from the beginning of a disk page. This storage layout is called the N-ary Storage Model (NSM). As shown in Figure 2.1, NSM might uses an offset table at the end of the page to locate the beginning of each tuple, if the tuple size is variable. Moreover, each tuple starts with a tuple header (RH) containing information about the tuple. RH starts with a null bitmap to support null values, offsets to the variable-length values to support variable attributes, and other implementation-specific details to make the layout more flexible.

2.1.2 DSM

While being a good fit for random-access algorithms, the NSM model suffers from two performance issues (16). First, since there is no spatial locality between attributes, NSM

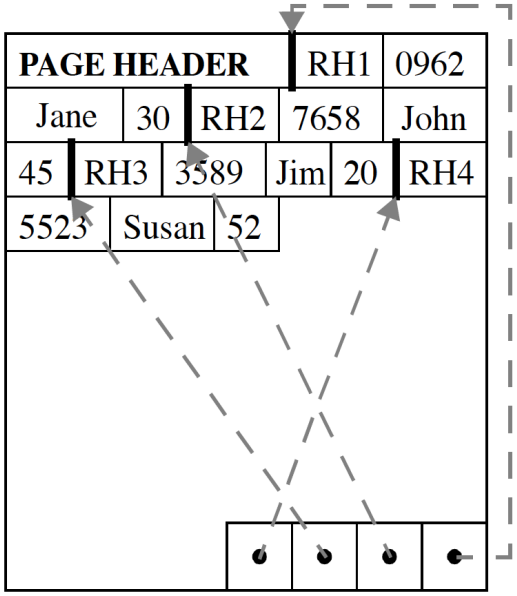


Figure 2.1: An example of an NSM page (this figure is borrowed from the paper by Ailamaki *et al.* (1))

incurs many data cache misses. A common example is when during the scan operation only one attribute needs to be processed, but NSM fills the cache with non-referenced attributes, resulting in poor cache performance. Second, if only a few attributes need to be loaded into memory, NSM introduces I/O bandwidth bottlenecks as many non-required data needs to read. To solve these problems, Copeland *et al.* propose the Decomposition Storage Model (DSM), where data is stored column by column (28). In contrast to NSM, DSM provides only those attributes that are needed. Also, DSM provides more efficient cache utilization as attribute values are clustered together. Finally, compared to NSM, DSM provides more compression opportunities because of similarity of adjacent tuples (29).

2.1.3 PAX

While DSM is a better model for Online Analytical Processing (OLAP) (30) workloads, if the entire tuple needs to be reconstructed, DSM incurs performance overhead as tuple reconstruction needs to be done explicitly at run-time. Ailamaki *et al.* propose Partition Attributes Across (PAX) (1), a technique that combines the inter-tuple spatial locality of DSM with low tuple reconstruction cost of NSM. PAX stores all attributes of a tuple inside the same page. However, within a page, PAX stores all attributes column by column

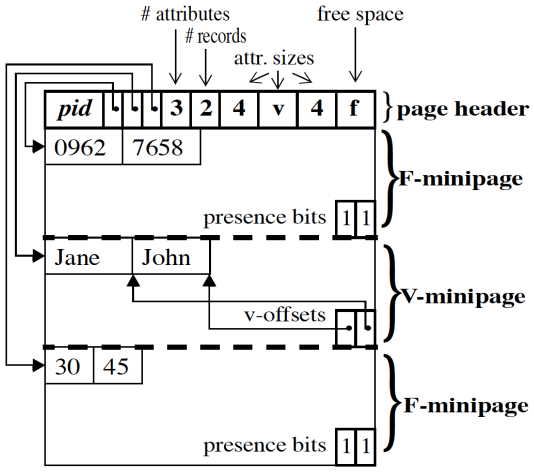


Figure 2.2: An example of an Pax page (This Figure is borrowed from the paper by Ailamaki *et al.* (1))

inside a *mini-page*. A mini-page is a partition of a page that stores all attributes inside a column sequentially and has two layouts two layouts. A fixed-length (F-minipage) to support fixed-length attributes, which also has a bit map located at the end denoting null values and Variable-length (V-minipage) to support variable-length attributes. As shown in Figure 2.2, each PAX page header contains the size and number of attributes, offsets to the beginning of each mini-page, the total number of tuples and the total space available.

2.2 Bit-(Un)Packing

In a 32-bit system architecture, integers are typically stored using 32 bits, while it might be possible to store them using fewer bits. For example, a 32-bit integer with value 70 (0b1000110) can be stored using 7 bits instead of 32 bits. In general, integers in the range $[0, 2^b]$ can be encoded using b bits and concatenated together into a single bit string. This process is called bit-packing. The reverse operation, i.e, the transformation of a bit string back into an array of machine-addressable integers is called bit-unpacking. Bit-unpacking can be implemented using five simple operations (load, shift, and, or, and store) to extract each integer as shown in Figure 2.3. After loading a 32-bit data into the CPU register, the desired bits are placed at the beginning of the register using the right-shift operation. Furthermore, the bit-wise AND operation with the predefined bitmask register (filled with 0 for random bits and 1 for desired bits) changes the bits with arbitrary values to 0. This implementation is efficient since it does not involve any branching. Zukowski *et al.* propose

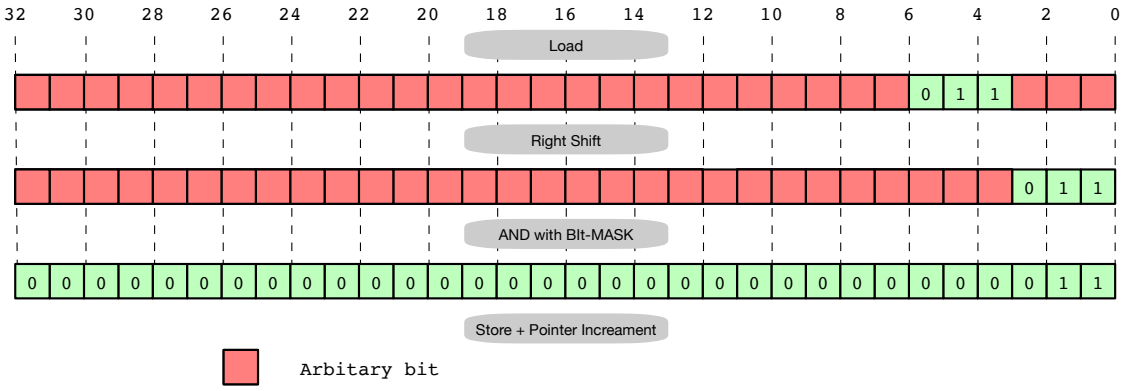


Figure 2.3: An example of Bit-Unpacking

implementing bit-unpacking for each bit-width as a loop-unrolled routine that bit-unpacks 32 integers in each iteration (31). This results in an always 32-bit aligned layout, as it requires bit-packing of 32 consecutive integers.

2.2.1 Vectorized Bit-Unpacking

Typically, bit-packing compresses integers in a sequence according to their original order. Schlegel *et al.* call the resulting compressed layout horizontal (2). Willhalm *et al.* propose a novel SIMD approach to bit-unpack horizontal compressed data layouts (32). Their approach consists of three sequential steps:

1. 16-Byte Alignment: 128-bit data is loaded from memory using one 128-bit SIMD aligned load instruction. For some bit-width values, such as 9, the first integer value might not be entirely loaded as it spans between two 128-bit registers. To handle these cases, the authors prefer to use one 128-bit SIMD register-concatenate instruction instead of 128-bit SIMD unaligned load instruction as SIMD unaligned instruction was expensive in terms of computation on older processors.
2. 4-Byte Alignment: four compressed integer values are copied to four separate 32-bit lanes in a new register using a SIMD shuffle mask instruction.
3. Bit Alignment: aligns all four integers in the last register at the first bit of their corresponding lane by shifting each lane an arbitrary number of times. This step is simulated via two instructions: a SIMD multiplication by four different integers and a SIMD right shift.

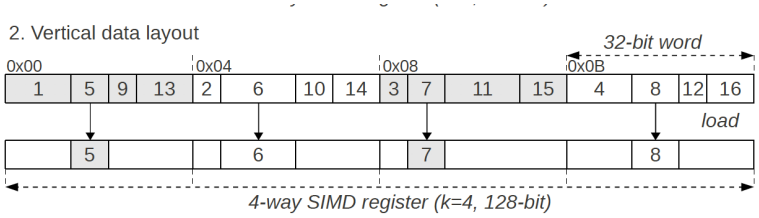


Figure 2.4: 4-way Vertical layout (This figure is borrowed from the paper by Schlegel *et al.* (2))

Willhalm *et al.*'s approach, however, only applies to SSE and cannot be extended to AVX2 as the shuffle instruction cannot move bytes cross all lanes. Polychroniou *et al.* propose a change to tailor Willhalm *et al.*'s approach for AVX2 (33). To simulate a 256-bit cross lane shuffle, we can copy each half (128 bits) of a register to both halves and use a regular 256-bit shuffle. Also, Willhalm *et al.* translate their bit-unpacking implementation to Intel AVX2 instructions (34). In Willhalm's new implementation, the simulated shift operation is replaced by an AVX2 vector-vector shift instructions. Furthermore, contrary to their previous implementation, the new implementation uses the unaligned load instruction instead of aligned load, as they claim that for current architectures, this instruction becomes even faster than aligned load. There is a performance penalty only for cases where data loads are split across cache lines, which is amortized by the reduction in other cases.

In addition to the horizontal layout, wherein tuples are stored successively, Schlegel *et al.* propose an alternative layout called vertical (2). In a $k-$vertical layout, each of the k consecutive bit-packed tuples is stored in a different memory word. Figure 2.4 shows an example of a 4-way vertical layout where each number indicates an integer's position in a sequence. As can be seen, every 4 consecutive integers are distributed among 4 different words. The vertical layout enables us to load/store compressed data using a single load/store SIMD instruction, without using the permutation instruction to distribute the tuples into the SIMD lanes. Lemire *et al.* use a 4-way vertical layout to bit-pack 128 tuples using the same bit-width to take advantage of SSE SIMD instructions (35).

2.2.2 BitWeaving

Willhalm *et al.*'s SIMD bit-unpacking approach has two limitations (32). First, for running a scan on encoded data, data needs to be first bit-unpacked. Note that it is possible to avoid the third step of bit-unpacking discussed in Section 2.2.1. Bit-unpacked data, in

contrast to bit-packed data, does not utilize all bits of a word. For example, if 9 bits are used to compress data, three values can be loaded inside a 32-bit register and 5 bits are wasted $(32 - (3 * 9) = 5)$, compared to only one for uncompressed data where 23 bits are wasted $(32 - 9 = 23)$. Therefore, bit-unpacking potentially increases the number of instructions required to process data. Second, Willhalm *et al.*'s approach wastes multiple cycles to align data, which could be avoided.

Li *et al.* propose BitWeaving, a technique to fully utilize the entire width of the processor words (or SIMD register) to reduce the number of instructions required to scan data (36). In Bitweaving, the scan operator compares a sequence of tuples with a given constant based on a comparison condition that is already defined and outputs a bitmap in which the corresponding bit is 1 if the tuple satisfies the comparison condition or 0 otherwise. Bitweaving offers two storage layouts: Horizontal Bit-Parallel (HBP) and Vertical Bit-Parallel (VBP). The combination of HBP and VBP is called BitWeaving, which comes in two flavors: BitWeaving/H and BitWeaving/V. BitWeaving/H is based on the HBP storage layout, while VBP is based on a combination of both.

In HBP, a column is divided into fixed-length segments that contain $(k+1) \cdot \lfloor \frac{w}{k+1} \rfloor$ tuples where w is the width of a processor word and k is the maximum number of bits needed for a tuple. In case of a CPU with SIMD capabilities, w can be as wide as the SIMD register. Each tuple is packed using $k+1$ bits, k for the tuple and one extra bit as delimiter. Inside the processor word, $\lfloor \frac{w}{k+1} \rfloor$ tuples are concatenated together and padded with 0s up to the word boundary. The scan operation for the HBP storage layout:

1. Generates a processor word Y by concatenating the constant provided by the scan operator, $\lfloor \frac{w}{k+1} \rfloor$ times.
2. Compares all X_is with Y and outputs a bit vector Z_i indicating whether the tuples in X_i satisfy the comparison condition. All comparison conditions are implemented as a function using simple instructions such as logical AND, logical OR, exclusive OR, binary addition, negation, and k-bit left or right shift. All SIMD extensions support these operations.
3. Shifts i times the output bit vector Z_i to the right, and combines it with the previous result.

IN VBP, a column is divided into fixed-length segments that contain w tuples, where w is the width of a processor word. In case of a CPU with SIMD capabilities, w can be as wide as a SIMD register. Considering k as the maximum number of bits required to represent any tuple in a column, a segment consists of k w-bit words, denoted as v_1, v_2,

..., v_k, such that the j-th bit in v_i is equal to the i-th bit in the original tuple c_i. The scan operation for the VBP storage:

1. Creates a list of words $C_1, ..., C_k$ to represent a constant C, such that all bits of C_i are 1 if i-th bit of C is 1. Otherwise, all bits of C_i are 0.
2. Iterates over all the segments and for each comparison condition applies the corresponding algorithm. For details of the algorithms please refer to the original paper by Li *et al.* (36).

For comparing a tuple against a constant, we can start from the most significant bit to the least significant bit until we find two different bits. After seeing the first pair, there is no need to continue the comparison. VBP enables early pruning, which is based on the idea of having direct access to the most significant bits of K words. Therefore, we can continue scanning until we find different bits for all w bits instead of comparing all the bits. Furthermore, the VBP format could be combined with HBP to avoid wasting memory bandwidth during early pruning.

While offering a fast scan with early pruning possibilities, the BitWeaved layout puts a considerable burden on any other operation than scan because of the high tuple reconstruction overhead. Polychroniou *et al.* propose a SIMD approach to mitigate this problem (33). Polychroniou *et al.*'s approach uses multiple SIMD registers to hold the unpacked words and continues distributing bits of Bitweaved layout among SIMD lanes of these registers till all tuples are reconstructed. This approach could be optimized using 8-bit lanes instead of 32-bit lanes. The performance of the optimized algorithm implemented for AVX2 is $18b + 30$ SIMD instructions for 64 tuples.

2.3 Light-Weight Compression Algorithms

Compression algorithms are divided into two categories based on their decoding speed: *Heavy-weight* methods and *Light-weight* methods. Heavy-weight compression algorithms such as Huffman (37), arithmetic encoding (38), and variants of the Lempel-Ziv algorithm (39) sacrifice the decoding speed for better compression speed. In contrast, light-weight methods such as Frame-of-Reference, RLE, and Dictionary encoding are faster schemes in decompression but have lower compression ratios. Abadi *et al.* show that light-weight methods could achieve an even better comparison ratio in the column-oriented databases as adjacent tuples are stored near each other (29).

2.4 Frame of Reference

Goldstein *et al.* propose Frame of Reference (FOR), a new compression algorithm tailored for database applications (40). This approach is particularly effective for tuples with many low to medium cardinality and numeric fields, and supports fast decompression. One of the most important differences of Goldstein *et al.*'s work compared to previous algorithms, such as Lempel-Ziv (41), is that the approach presented by Goldstein *et al.* can decompress individual tuples rather than the full page or an entire relation. This algorithm is based on the observation that the actual range of values that appear in a given column on a given page is much smaller than the range of values in the underlying domain. For each page, FOR finds the maximum M and the minimum m value of a numeric column, subtracts m from all values and stores all using $\lceil \log_2(M + 1 - m) \rceil$ bits each. FOR only needs $\lceil \log_2(M + 1 - m) \rceil$ bits to store values instead of $\log_2(M)$ bits, which is typical in other algorithms such as simple bit-packing. This results in compression ratios of 4 to 1 and 88 to 1 on real and low-cardinality datasets, respectively (40). Table 2.1 gives an overview of all variants of FOR. In the following we discuss each variant in more detail.

2.4.1 PFOR

FOR, while providing fine-grained access and better decompression speed compared to previous work, still is vulnerable to outliers, which are very common in practice. Zukowski *et al.* propose Patched Frame of Reference (PFOR) to alleviate the vulnerability of outliers (31) (42).

PFOR uses a single bit-width b for the entire page. The value b is chosen by gathering samples and then selecting the one that yields the best compression ratio. PFOR compresses and decompresses each 128 input values together, which we refer to as chunk, and classifies each value as either *exception* or *coded*. An input value is an exception if it is higher than 2^b (`MAXCODE`) and will be stored on the disk as an offset pointing to the next exception in the chunk of 128 integers. The actual values for exceptions are stored in a separate location called the *exception section*. If an integer is less than `MAXCODE`, it is a coded value and is stored as a regular integer. All values, either coded or offset, are bit-packed and stored in a section called the *code section*. In addition to these two sections, there is another section designed to provide fine-grained access by storing a pointer to the first exception value of each chunk and their corresponding value in the exception section. Note that if the gap between two exceptions is higher than `MAXCODE`, a regular value in

	chunk size	bit-width	base	compulsory	exceptions	compressed exceptions
PFOR	128	per page	per page	yes	per page	no
PFOR2008	128	per page	per page	yes	per page	8, 16, 32 bits
AFOR-1	32	per chunk	per page	no	no exception	no exception
AFOR-2	8,16,32	per chunk	per page	no	no exception	no exception
AFOR2-3	8,16,32	per chunk	per page	no	no exception	no exception
NewPFD	128	per chunk	per page	no	per chunk	Simple-16
OptPFD	128	per chunk	per page	no	per chunk	Simple-16
SIMD-FASTPFOR	128	per chunk	per page	no	per page	vectorized bin. pack
FASTPFOR	32	per chunk	per page	no	per page	binary packing
SIMPLEPFOR	32	per chunk	per page	no	per page	binary packing
S4-FASTPFOR	128	per chunk	per page	no	per page	binary packing
C-PFOR	1024	per chunk	per chunk	yes and no	per page	Composable compression

Table 2.1: Overview of all variants of FOR

the middle needs to be treated as an exception (called a compulsory exception) to keep the exception list connected.

To summarize, PFOR treats outliers as exceptions such that the minimum and maximum range for an integer column is greatly reduced. Therefore, PFOR can achieve a better compression ratio compared to FOR. Additionally, the PFOR implementation does not use conditional branches, such as if-then-else constructs in performance critical parts of its compression and decompression routines to achieve high Instructions Per Cycle (IPC) efficiency for super-scalar CPUs. PFOR reports decompression speed in the range of greater than 2 GB/s which is an order of magnitude faster than conventional compression algorithms such as LZRW1 (43), the fastest version of the Lempel-Zip compression.

Zhang *et al.* introduce two changes to the implementation of PFOR to improve the compression ratio. The extended algorithm is known as PFOR2008. Firstly, the authors use 8, 16 or 32 bits per exception instead of always using 32. Secondly, Zhang *et al.* allow any value of b. We should note that the original PFOR method allows any value of b and uses 8 just for a prototype. Therefore, the second change is not actually an improvement over PFOR.

2.4.2 AFOR

Delbru *et al.* propose Adaptative Frame of Reference (AFOR), a scheme with the same goal as PFOR but different storage layout (44) (45). Compared to PFOR, AFOR does not treat outliers as exception. Instead, AFOR partitions a page into multiple frames containing 8, 16, or 32 tuples. Each frame has a separate Bit Frame Size (BFS) stored before the frame using 8 bits. Choosing an optimal frame size and BFS can be solved using Dynamic Programming algorithms (46) but this would incur a considerable computation overhead. Delbru *et al.* instead use a local optimization algorithm to have a satisfactory compression, which is also efficient to compute. Their local optimization algorithm reads the next 32 values, which can be one of these combinations: $[32]$, $[16; 16]$, $[16; 8; 8]$, $[8; 16; 8]$, $[8; 8; 16]$ or $[8; 8; 8; 8]$. For each combination, the bit frames for all frames and then the compressed size is calculated. After that, the combination with the smallest size is chosen. Delbru *et al.* report that compared to PFOR, AFOR provides a similar compression ratio on frequency inverted files, but performs much better on document and position files. In terms of decompression speed, the authors report that PFOR is slightly faster then AFOR.

It is also worth mentioning that there are three variants of AFOR, namely AFOR-1, AFOR-2, and AFOR-3. In AFOR-1, the frame size is always 32, while in AFOR-2 it can be 8, 16 or 32. AFOR-3 is similar to AFOR-2 but employs the *frame stripping* technique.

If all 32 values inside a frame are 1, the frame stripping technique stores a special BFS and avoids storing 32 bits with value 1, which saves 32 bits.

Note that Anh *et al.* call techniques such as AFOR-1, where data is compressed by applying bit-packing to a group of integers, PackedBinary (47), whereas Lemire *et al.* refer to them as binary packing (48)(35). PackedBinary has only a small difference compared to AFOR-1, that the frame size is arbitrary. Anh *et al.* have found 16 is the best frame size.

Finally, To SIMDize AFOR-1, Lemire *et al.* propose S4-BP128, which an arbitrary bit-width is chosen for every 128 values, and the data is bit-packed using a 4-way vertical layout (48).

2.4.3 VSEncoding

Silvestri *et al.* propose Vector of Splits Encoding (VSEncoding), which in contrast to AFOR, finds the optimal frame size using dynamic programming (49). As a result, VSEncoding provides 10% better compression ratio than other schemes. Also, to provide fast decompression, all frames of integers with the same bit-width are stored together.

2.4.4 NewPFD and OptPFD

As mentioned in Section 2.4.1, if two consecutive exceptions have a distance of more than 2^b, PFOR is forced to use compulsory exceptions, which implies extra overhead to save a regular value as an exception. To avoid compulsory exceptions, bit-width could be increased. This results in even a worse compression ratio as it also increases the number of bits used for all codes. Yan *et al.* propose two new versions of the PFOR, namely, NewPFD and OptPFD, to overcome this problem (50). Yan *et al.* introduce a new structure for PFOR, which

- Determines the b value for each chunk instead of the whole page.
- Stores the lower b bits of an exception instead of the offset to the next exception.
- Stores the higher bits of an exception in the exception section instead of the entire value.
- Stores the offset values in a different section, which comes after each chunk. The chosen b value for this chunk will be stored at the beginning of this section.
- Compresses both the offset section, and the exception section for each chunk.

This new approach is called NewPFD, which is similar to PFOR. NewPFD selects b in a way that the number of exceptions does not exceed a threshold, e.g., 5% of exceptions. The second proposed version of PFOR is OptPFD, which in contrast to NewPFD, models the selection of b as an optimization problem. The smallest compressed size for each chunk is originally assigned to b. Then the decoding speed is increased as desired by selecting a block that gives us the most time savings for increase in size, which is then assigned to b.

2.4.5 ParaPFOR

As mentioned in Section 2.4.1, PFOR and all of its variants are designed and tailored to perform best on CPUs. For example, in PFOR, decompression must be done serially because an exception will be encoded as an offset to the next exception. Thus, PFOR has poor performance on the Graphic Processing Units (GPUs). Ao *et al.*, in an attempt to improve the performance of PFOR on GPUs, propose a new version of PFOR called Parallel PFor (ParaPFor) (51). ParaPFOR stores indices of exceptions instead of a pointer to the next exception in the code section. This enables the GPU to concurrently recover exceptions. This modification leads to a worse compression ratio, but much faster decompression on the GPU.

2.4.6 SIMD-FASTPFOR, FASTPFOR and SIMPLEPFOR

NewPFD and OptPFD have two main differences compared to PFOR. PFOR stores exceptions on a per page basis, while NewPFD and OptPFD store exceptions on a per chunk basis. In addition, PFOR uses a single bit width for the entire page, whereas NewPFD and OptPFD choose a separate bit width for each chunk. The mentioned differences result in a better compression ratio, while the decompression speed degrades significantly (35).

Lemire *et al.* propose two new versions of PFOR called FastPFOR and SimplePFOR to combine the compression ratio of NewPFD and OptPFD and the decompression speed of PFOR. These two new compression algorithms store exceptions on a per page basis, similar to PFOR, while they choose a separate bit width for each chunk, similar to NewPFD and OptPFD. In NewPFD, for each chunk, the bit width, which minimizes the expected storage, is chosen. All the information needed for decompressing a chunk such as the bit width and the maximum number of bits used for exceptions is stored in a separate section which comes after the exception section. If the maximum bit width is greater than the number of allocated bits, a counter c indicating the number of exceptions and c exception

locations are stored. In contrast to NewPFD and OptPFD, which use compression, one byte is used to store each number in this section.

SimplePFOR and FastPFOR differ in how they compress high bits of exception values in the exception section. In the SimplePFOR scheme, all high bits of the exception values will be compressed using Simple-8b (47), while in the FastPFOR scheme, the exception section is divided into 32 arrays, one for each possible bit[1]. All exceptions are stored in one of the 32 arrays, from 1 to 32. The difference between the maximum bit width and the number of allocated bits determines in which array the exceptions are stored. Each of the 32 arrays is then bit packed using the corresponding bit width. Therefore each array length needs to be a multiple of 32. these arrays are padded if an array length is not a multiple of 32, which slightly diminishes the compression ratio.

SIMD-FastPFOR is identical to FastPFOR except that it uses vectorized bit-packing (see Section 2.2.1). Because of using vectorized bit-packing, the 32 exception arrays length should be multiples of 128, instead of 32, which degrades the compression ratio.

2.4.7 S4-FASTPFOR

Lemire *et al.* propose S4-FASTPFOR, which is the combination of SIMD-FastPFOR and FASTPFOR (48). Similar to SIMD-FastPFOR, S4-FASTPFOR uses vectorized bit-unpacking. In contrast to SIMD-FastPFOR, bit-packed exceptions are padded to multiples of 32. The S4-FASTPFOR performance is the same as SIMD-FastPFOR, while the compression ratio is improved by 5%.

2.5 Delta Coding (Differential Coding)

Ng *et al.* propose Delta Coding, also known as Differential Coding, a compression technique that compresses a sequence of integers by replacing each integer by its difference with respect to its preceding integer (52). If the differences between successive integers in a column are small, differential coding might be a better choice than other compression schemes. Since differences between successive values are small, fewer bits can be used to represent each integer, resulting in a better compression ratio. Also during the decompression phase, the original integer could be recovered from the deltas by one fast simple add operation, which makes delta coding a good candidate for database systems. However,

[1]SimplePFOR and FastPFOR are designed for integer values, therefore the maximum number of bits is 32

the downside of delta coding is that each integer value can only be calculated after the value of the integer immediately preceding it is known.

2.5.1 D4, DM, D2, D1

Lemire *et al.* propose four different forms of differential coding, namely, D1, D2, DM and D4. We explain these four methods with a simple example. If we consider 8 integers, $x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8$, deltas calculated using original delta coding are as follows:
$x_1, \delta_1 = x_2 - x_1, \delta_2 = x_3 - x_2, \delta_3 = x_4 - x_3, \delta_4 = x_5 - x_4, \delta_5 = x_6 - x_5, \delta_6 = x_7 - x_6, \delta_7 = x_8 - x_7$.
To recover original values, 7 sequential add operations are needed as follows:
$x_1, \delta_1 + x_1, \delta_2 + x_2, \delta_3 + x_3, \delta_4 + x_4, \delta_5 + x_5, \delta_6 + x_6, \delta_7 + x_7$
In D4, deltas are calculated four-by-four as follows:
$x_1, x_2, x_3, x_4, x_5 - x_1, x_6 - x_2, x_7 - x_3, x_8 - x_4$
which enables us to use a single 1-cycle-latency SIMD instruction to recover them. Deltas in DM are calculated as follows:
$x_1, x_2, x_3, x_4, x_5 - x_4, x_6 - x_4, x_7 - x_4, x_8 - x_4$
DM, compared to D4 requires one extra SIMD instruction, which is fast on Intel processors, to copy the last component. Delta calculation for D2 is as follows:
$x_1, x_2, x_3, x_4, x_5 - x_3, x_6 - x_4, x_7 - x_5, x_8 - x_6$
D2 can be implemented using 4 SIMD instructions (shift, add, select and add). D1 is similar to the original Delta coding but with a SIMD implementation which uses 6 SIMD instructions (shift, add, shift, add, copy and add). In terms of speed, D4 is the fastest and D1 the slowest. In terms of compression ratio, D4 has the worst compression ratio while D1 has the best one mainly because deltas are smaller (we assume that successive integers are close to each other, therefore $x_4 - x_1$ is bigger than $x_2 - x_1$, which needs more bits to be represented).
Furthermore, to handle outlier deltas, Lemire *et al.* combine D4, DM, D2, and D1 with S4-BP128 resulting in S4-BP128-D4, S4-BP128-D3, S4-BP128-D2, S4-BP128-D1 (48), respectively. Similar to S4-BP128, all these new schemes bit-pack every 128 deltas using an arbitrary bit-width.

2.5.2 Vectorized Prefix Sum

As explained in Section 2.5.1, D1 could be implemented using 6 SSE SIMD instructions. This implementation is dependent on the SIMD shift instruction that shifts all bits together

arbitrary times. However, this instruction does not exist for wider registers. Zhang *et al.* propose extending this implementation to AVX-512 by simulating the 512-bit shift instruction with two SET and ALIGNR instructions (53). The resulting implementation, called *horizontal*, needs 12 instruction for every 16 integers. Moreover, Zhang *et al.* propose two different SIMD prefix sum computation techniques, *vertical* and *tree*, which we do not discuss in this related work as they are slower than horizontal.

2.5.3 PDelta and PFOR-Delta

In the D1 compression layout, the first integer is intact, which might have an adverse effect on the compression ratio as this integer increases the minimum number of bits needed to represent deltas. PDelta introduces a base parameter equal to the first integer, so that the first integer can be encoded as 0. Furthermore, PDelta uses the patching technique discussed in Section 2.4.1 to handle outliers. To further optimize the compression, PFOR can be applied on deltas, called PFOR-Delta (31).

2.5.4 V-PFORDelta

Compression schemes such as PFOR, which use b bits to encode most of the data and store exceptions separately are called patched encoding. In the patching phase, the exception values are replaced by their original position. In case of PFOR, the offset to the next exception is stored instead of an exception, while in FastPFOR, the first b low bits of an exception. None of the related work discussed in this section are able to vectorize the patching phase. Al Hasib *et al.*, in an attempt to vectorize patched encoding, introduce V-PFORDelta (54). V-PFORDelta, inspired by D4 delta coding, stores a bitmap for each chunk to determine which integer in each chunk is an exception. All exceptions are replaced with 0, while their values are stored continuously in a separate location. During decompression, the exceptions are placed in their right position inside a SIMD register by using a SIMD unpacklo instruction (We note that the shuffle instruction should have been used instead of unpacklo). After that, with a simple SIMD add operation, the original values can be recovered.

2.6 Dictionary Encodings

Dictionary Encoding is a lossless compression technique that encodes values as integer codes that map to the values. It is useful when a column is composed of value distributions

that only use a subset of the full domain. Furthermore, integer codes could be compressed using bit packing.

2.6.1 PDICT

Similar to FOR, discussed in Section 2.4, dictionary encoding is vulnerable to outliers. To solve this problem, Zukowski *et al.* propose PDICT, which treats outliers as an exception and handle them using the patched coding technique (31). Note that, each chunk potentially has a separate dictionary list, but by combining multiple dictionary lists, we could have a better compression ratio.

2.7 RLE

Run-length Encoding (RLE) is another type of compression scheme that is useful in column-oriented databases (29). RLE is fundamentally different from FOR, differential coding, and dictionary coding. While these compression techniques represent the original data as a sequence of small integers, RLE reduces the number of values required to represent the original data. RLE compresses runs of the same value in a sequence of integers into a pair of *Run* and *Length*. Run represents the value of the repeated integer and Length represents how often the value is repeated. For example the string 1111223355555555 can be compressed into 14223258, where each repeated run is replaced by the run and length values. Note that there are different variations of RLE, for example, we can store the start position of a run instead of its length.

2.7.1 Vectorized RLE

Damme *et al.*, in an attempt to vectorize the decompression phase of RLE, propose a new implementation which works as follows (55). The decompression module repeats the following steps until the entire input data has been consumed.

- Load the next pair of run value and run length.
- Load one SIMD register with copies of the run value.
- Store the contents of that register as often as required to match or pass the run length at *offset* (using the unaligned SIMD store instruction). Offset refers to the memory address where the result should be stored.
- Increase offset by run length.

This algorithm is correct since the next SIMD store operation overrides extra elements written by the previous SIMD store. Furthermore, this new algorithm could be adapted to all SIMD extensions such as SSE, AVX, AVX2, and AVX-512 registers because this algorithm is only dependent on SIMD unaligned store instruction, which is supported by all SIMD extensions.

2.8 Fast Static Symbol Table

Database systems often use LZ4 to compress blocks of strings. Although LZ4 provides a good decompression speed, it does not allow random access to compressed data as it is *block-based*: the complete block needs to be uncompressed before being ready for any further processing. To alleviate this problem, Boncz *et al.* propose FSST (56), a string compression scheme that allows random-access to compressed data while providing similar decompression speed to LZ4. FSST's compression is based on the idea that the strings of a column often have common substrings (called symbols, have a size of 1 to 8 bytes) that could be replaced with short 1-byte codes. FSST also represents less frequent substrings as exception, stored as a code with the value 255, followed by the original byte. During compression, symbols are identified and used to create a symbol table that maps codes to symbols. During decompression, each code is replaced by its corresponding symbol. Decompression is fast as it requires few instructions, is branch-free and cache efficient as both the symbol table and the length array easily fit into the level 1 CPU cache.

2.9 General SIMD-based Compression algorithms

Inspired by the 4-way vertical layout (see Section 2.2.1, Zhao *et al.* propose a general SIMD-based approach to accelerate compression algorithms (57). Their approach is based on the observation that often the storage layout of a compression algorithm consists of encoded data and a separate part which holds information about the encoded data. Therefore, data could be encoded in a 4-way vertical layout to enable the use of SIMD instructions. To show their approach's flexibility, the authors proposed new algorithms such as Group-AFOR and Group-PFD, which are the SIMD-based version of compression algorithms such as AFOR and PFD. Group-AFOR is very similar to SIMD-BP128, with one difference that in SIMD-BP128, the frame size is always 128, while in Group-AFOR, it is 32, 64, or 128. Group-PFD handles exceptions like Zhang *et al.*'s approach while encodes integers like the SIMD-BP128 approach. Group-PFD and Group-AFOR are benchmarked against the

state of art algorithms and the result shows they are almost as fast as the state of art algorithms.

2.10 Whitebox Compression

Ghita *et al.* call all existing compression techniques, including the ones we discussed so far, black-box compression techniques, as their decompression logic is hard-coded and query operators in databases cannot directly operate on compressed data (3). Ghita *et al.* propose white-box compression to makes (de)compression transparent and optimizable. The white-box compression model represents *logical* columns as composite *functions* of *physical* columns. Logical columns are defined by the database schema containing the tabular structure that the user expects to see, while physical columns are what is stored on disk. The functions are standard column expressions that are fast and expressive enough to handle different patterns. It is worth mentioning that white-box compression potentially provides a better compression ratio as multiple logical columns could be stored as fewer or more compresseable physical columns. For example, using white-box compression, the logical columns shown on the left side of Figure 2.5 can be stored as physical columns shown on the right side. In case of the logical column B, this is achieved by mapping each string to an integer. Thus we store integers and a dictionary on disk, and during runtime, the de-mapping function could recover original values. Column A could be represented by a function that concatenates a string, the symbol '-', and an integer together. The first string is generated by a function that abbreviates strings using a dictionary that is used for column B. Therefore, only integers need to be stored in an independent physical column. As can be seen, strings have been converted to low-cardinality integers, which enables us to use FOR instead of LZ4, resulting in a better compression ratio. As can be seen, strings have converted to low-cardinality integers, which enables us to use FOR instead of LZ4, resulting in a better compression ratio.

2.11 Big Data File Formats

In this section, we discuss the storage layout and the compression techniques used in the recent and state-of-the-art data file formats.

A	B	P	Q
"GSA_8350"	"GENERAL SERVICES ADMINISTRATION"	0	8350
"GSA_8351"	"GENERAL SERVICES ADMINISTRATION"	0	8351
"HHS_2072"	"HEALTH AND HUMAN SERVICES"	1	2072
"TREAS_4791"	"TREASURY"	2	4791
"TREAS_4792"	"TREASURY"	2	4792
"HHS_2073"	"HEALTH AND HUMAN SERVICES"	1	2073
"GSA_8352"	"GENERAL SERVICES ADMINISTRATION"	0	8352

Logical Physical

Figure 2.5: An example of Whitebox compression (This Figure is borrowed from the paper by Ghita *et al.* (3))

2.11.1 Parquet

Traditionally, column stores have been optimized to store relational data, not nested data models. To store nested data in columnar format, the structure of data also needs to be stored in one or more columns such that records could be stored in flat columns and read back to their original format efficiently. Melnik *et al.* propose a novel columnar storage format for nested data (58). In their proposed format, the structure of a record is captured for each value by two integers called *repetition level* and *definition level*. The repetition levels show the structure of nested types, while the definition level denotes whether the value is null.

In Melnik *et al.*'s format, each column is stored as a set of blocks containing the repetition and definition levels and compressed field values. Note that Definition levels and repetition levels are not stored if data is flat. Furthermore, NULLs are not stored explicitly as definition levels already determine them.

Parquet is a compressed, columnar storage format that uses Melnik *et al.*' approach to store nested data. As shown Figure 2.6 (left), a Parquet file consists of one or more *row groups* where a row group is a logical horizontal partitioning of data into rows. Moreover, a row group ensures that attributes within the same record are stored near each other. Each row group contains precisely one *column chunk* per column. A column chunk is a chunk of data for a particular column, which is divided into *pages* that are stored sequentially. As shown in Figure 2.6, a page consists of a page header serialized by Apache Thrift (59), repetition levels, definition levels, and the actual encoded data. As shown in Figure 2.6 (left), in a Parquet file, metadata containing information about the location and types of

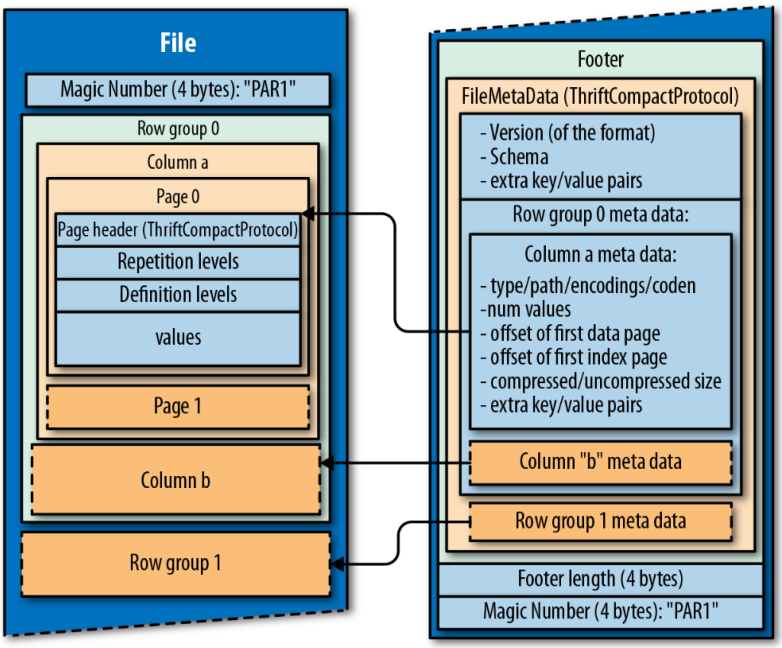


Figure 2.6: The Parquet file format

column chunks is serialized using Apache Thrift's Thrift Compact Protocol and is stored as a footer.

Moreover, Parquet is designed to be a flexible data format. For example, Parquet saves space by not storing the repetition or definition levels if data is not nested or nullable. Furthermore, row group and page size can be specified by the user before writing a Parquet file. Larger row groups allow larger column chunks, making it possible to do larger sequential IO, while requiring more buffering space during writing. Besides, larger pages allow avoiding the space and processing overhead of page headers, while making fine-grained access harder as the entire page needs to be fetched.

In terms of compression, Parquet offers the following encodings schemes[1]:

- RLE/Bit-Packing Hybrid: a combination of RLE and simple bit-packing. Depending on the characteristics of every 128 consecutive values (chunk) inside a page, Parquet chooses one of these two methods. Note that all chunks compressed using bit-packing have the same bit-width.
- Dictionary Encoding : Simple Dictionary encoding where integers codes are further compressed using RLE/Bit-Packing Hybrid.

[1] `https://github.com/apache/parquet-format/blob/master/Encodings.md`

- Delta Encoding: Similar to S4-BP128-D1, where deltas are calculated as the differences between consecutive elements and the smallest delta is substraced from them to guarantee that all values are non-negative.
- Delta-length byte array: Parquet stores a byte array as a combination of 4-bytes length and the actual bytes. To compress a stream of byte arrays, the length of each element is extracted and compressed using Parquet delta encoding and further concatenated to the end of the byte array stream.
- Plain: If none of the above schemes is detected suitable for given data, Parquet stores data in the plain format.

2.11.2 ORC

Besides Parquet, another popular big data file format is Optimized Record Columnar (ORC) which is proposed by Huia *et al.* (4). ORC is a columnar file format designed to address the shortcomings of RC file format (60). As shown in Figure 2.7, an ORC file consists of one or more stripes, file footer, and postscript.

stripes: ORC, first, horizontally partitions a table to multiple stripes. A stripe is similar to a row group wherein data is stored column by column. A portion of a column, stored inside a stripe, is called a stream. Together with each stream, the metadata of a column is stored, called a metadata stream, which is useful to analyze which stripes are needed to evaluate a query. Furthermore, data values inside each stream are divided into fixed-sized groups called an index group (the default index group size is 10000). For each index group, data statistics of those values are also recorded. Besides the actual data, a stripe is consists of the index data and a stripe footer. The index data keeps the starting points of every index group in metadata streams and data streams as a column in a stripe might have multiple logical index groups that should be known to the reader of the ORC file format. The stripe footer contains a directory of stream locations.

File footer: The file footer includes data statistics of each column which are useful for query optimizations and aggregation queries. Also, the file footer keeps the starting point of each stripe.

Postscript: postscript holds compression parameters and the size of the compressed footer.

In terms of compression, ORC uses a two-level compression scheme. First, a stream is encoded based on its characteristic using one of the four types of encoding: 1) a sequence of bytes, 2) a sequence of RLE encoded bytes, 3) a sequence of integers encoded by RLE

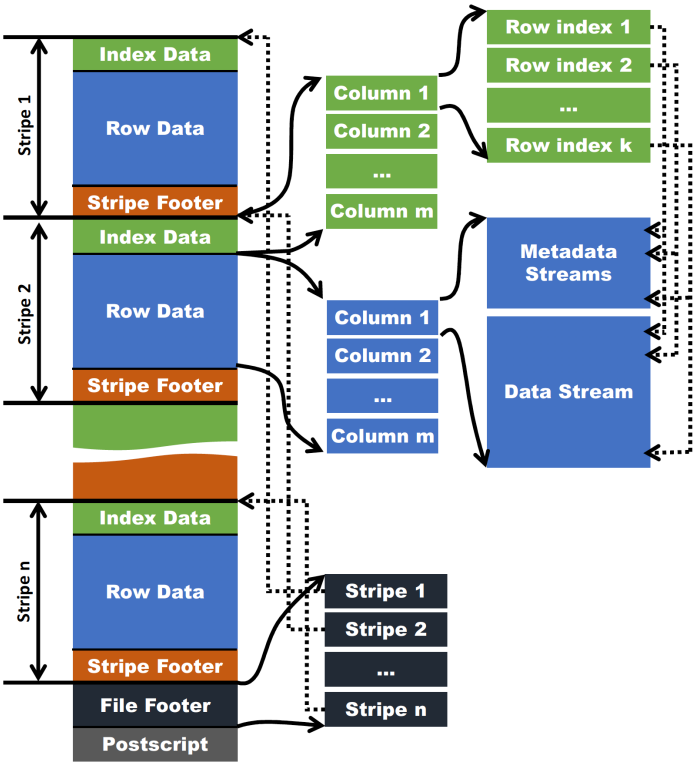


Figure 2.7: The ORC file format (This Figure is borrowed from the paper by Huai *et al.* (4))

or delta encoding and 4) a bit vector. Then, the result is further compressed using an optional general-purpose data compression scheme such as ZLIB, Snappy, or LZO.

2.11.3 Artus

Chattopadhyay *et al.* propose a new columnar file format called Artus, which is designed to support both fast row lookups and range scans (61). To achieve fast lookups, Artus uses encoding methods that allow random access to compressed data before decompression, and for sorted columns, only uses encoding methods that allow binary search.

Furthermore, Artus proposes a novel representation for nested data types. Considering the table's schema as a tree of fields, Artus store a separate column on disk for each field, unlike Parquet, that stores only leaf fields. This representation enables Artus to store sparse data more efficiently then Parquet as it does not record any information about fields whose parent does not exist. Moreover, as discussed in Section 1.2, Artus tries to expose encoding information to the evaluation engine.

2.11.4 Data Blocks

Lang *et al.* propose a new compressed columnar storage format called Data Blocks, which differs from other storage formats as it does not use bit-packing (5). Based on the assumption that current bit-unpacking methods are not fast enough, even those with a SIMD implementation, Data Blocks stores tuples in a byte-addressable compressed format (A tuple is compressed into a 1-, 2-, or 4-byte integer).

Data Blocks stores one or more attributes of up to 2^{16} tuples in a self-contained container, called Data block. Note that in case of storing all attributes, the Data block resembles the PAX storage layout (1). As shown in Figure 2.8, a Data Block contains a tuple counter, and for each attribute contains the compression method, offsets to the attribute's Small Materialized Aggregates (SMA), dictionary, compressed data vector, string data, SMA, PSMA, and the actual data.

Moreover, Data Blocks compress an attribute inside a Data Block using one of the following techniques:

- Single value compression: a special case of run-length encoding when all values of an attribute in a block are equal or NULL.
- Ordered dictionary compression: a special case of dictionary encoding where if for two uncompressed values k_i and k_j, $k_i < k_j$ holds, then for their dictionary-compressed values dk_i and dk_j, $dk_i < dk_j$ holds.

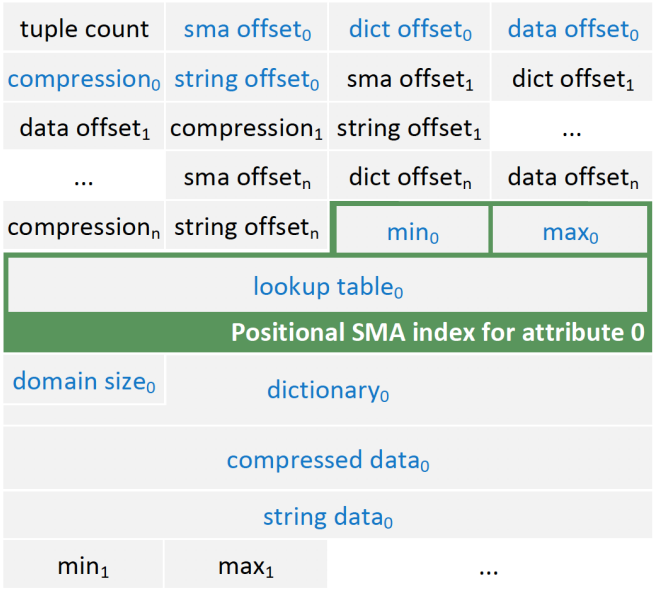


Figure 2.8: Layout of a Data Block for n attributes (This figure is borrowed from the paper by Lang *et al.* (5))

- Truncation: a special case of FOR which avoid bit packing and the base is equal to the attribute's minimum value.

Note that the choice of the compression scheme depends on the attribute's value domain in the specific block. Thus, different ranges of an attribute might be compressed via various schemes.

2.11.5 Albis

Trivedi *et al.* propose an entirely new big data file format called Albis (62). Based on the assumption that CPU is the new bottleneck for big data processing, Albis tries to reduce CPU pressure as much as possible. Albis achieves this by storing data in binary format with no compression, avoiding unnecessary object materialization, and separating the metadata from the actual data.

Albis split a table into multiple vertical groups called column-group (CG), where each group can have one or many columns. If a CG contains only one column, the format is similar to DSM, while in case of more than one column, it is similar to a row-store. Besides vertical partitioning, Albis partitions a table horizontally to multiple row groups.

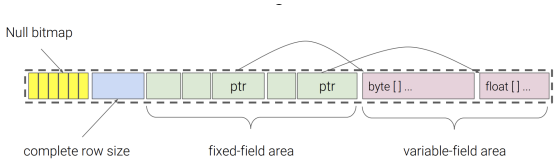


Figure 2.9: An example of a Row format in Albis

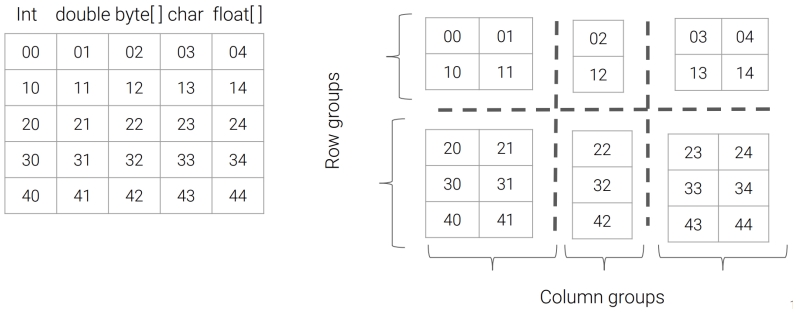


Figure 2.10: An Example of table partitioning in Albis

Figure 2.10 shows an example of a table partitioned by Albis. Inside a CG, data is stored row by row, where each row consists of:

- A size counter showing the total spaced used by this row.
- A null bitmap denoting whether a value is null or not.
- A fixed-length section storing the actual data of columns. Columns with fixed-length data are stored as the way they are while variable-length columns are encoded by an 8-byte integer consisting of the offset and the size of variable-length fields, which are stored later in the variable-length section.
- A variable-length section storing columns with variable-length data.

Figure 2.9 shows an example of CG with five integer, double, byte array, char and float array columns. As a result, Albis consumes eight times more space compared to Parquet, while it speeds up the performance on TPC-DS queries three times.

2.12 Matrix Tranpose

Later in this thesis, we implement a Whitebox function that applies matrix transpose on uncompressed data to correct their order in a column. In this section, we review the

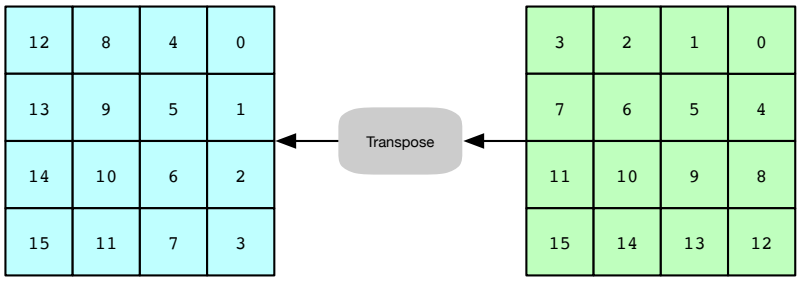


Figure 2.11: An example of the matrix transpose

recent studies that have been done to improve the matrix transpose performance. Matrix transpose is an operator which interchanges the rows and columns of a matrix. Figure 2.11 shows an example of a $4 * 4$ transposed matrix. Zekri *et al.* propose an algorithm for transposing a matrix of size $n * n$ using n-integer wide SIMD registers (63). For example, using this algorithm, an $8*8$ matrix can be transposed using AVX registers. However, this algorithm requires $3 * n$ registers, which can lead to register pressure for wider registers as for AVX-512, 48 registers are needed, but Intel AVX-512 has only 32 registers. Also, the complexity of this algorithm is $O(7n)$. Limonova *et al.* try to extend this algorithm to support 128-bit ARM NEON, which is outside of the scope of this thesis as this algorithm does not work on Intel CPUs (64).

3

SIMD-Friendly Bit-(Un)Packing

In this chapter we present our 1024-bit interleaved bit-(un)packing technique. As discussed in Section 2.2.1, the most efficient Bit-(un)packing approach is proposed by Lemire *et al.* (35), which uses a 4-way vertical layout. There are, however, three problems in using Lemire *et al.*'s layout in a file format:

- It is designed for 32-bit integers and 128-bit SSE registers as four 32-bit integers fit inside an SSE register. However, this layout cannot be extended to support 8-bit, 16-bit, and 64-bit integers as four of these integers need 32, 64, and 256 bits SIMD registers.
- It is not future-proof as it is specifically designed for SSE registers and cannot exploit new wider AVX-512 registers that are already available.
- The provided implementation of the 4-way layout is not suitable to be used in a file format as it does not support scalar bit-unpacking for the interleaved layout. The reader API implementation for a file format should support all CPU architectures.

Our 1024-bit interleaved Layout solves all these problems:

- Our 1024-bit interleaved layout uses 1024 bits to distribute integers among SIMD lanes instead of a fixed number of lanes (4 in case of 4-way vertical layout). This change makes the new layout capable of supporting all integers with different sizes.
- Using 1024 bits to interleave data makes our format capable of using 1024-bit instructions such as 1024-bit SHIFT, 1024-bit AND, and 1024-bit OR which will be introduced in the future. Note that 1024 is chosen for prototyping, and given that there are 2048-bit registers (27), and even wider register will be released in the near

future, we are aiming for a 4096-bits (or 8192-bits) interleaved layout. 4096-bit Interleaved layout works exactly as our 1024-bit interleaved layout, except that we use 4096 bits, instead of 1024 bits, to interleave data.

- Our 1024-bit interleaved bit-unpacking is implemented for all SIMD register sizes and also scalar CPUs. Our 1024-Bit interleaved bit-unpacking routines are provided as a library that chooses the right implementation, based on the CPU architecture, at compile time.

3.1 Storage Layout

In our 1024-bits interleaved storage layout, 1024 bits are divided into L W-bit words, $w_1, w_2, ..., w_l$, where W can be 8, 16, 32, or 64, the number of bits used to represent an integer in the bit-unpacked format. L is equivalent to the number of SIMD lanes a 1024-bit register can have for a given type of integer.

We now describe how our 1024-bit storage layout works. First, B bits (B is the maximum number of bits required to represent an integer) of consecutive integers are extracted. Then, the 1024-bit interleaved layout distributes the bit-packed integers among L words of the 1024 bit layout in the following order: the ith bit-packed integer goes to the $\lfloor \frac{i}{L} \rfloor$th word, i.e., the 0th bit-packed integer goes to the 0th word, the 1st bit-packed integer goes to 1st word and so on until the $L-1$th word. The Lth bit-packed integer goes to 0th word and is appended to last bit-packed data of the 0th word. Note that if we were not using interleaving, the ith bit-packed integer would be appended to $(i-1)$th bit-packed integer inside the $\lfloor \frac{i*b}{W} \rfloor$th word.

Figure 3.1 shows an example of 96 32-bits integers, represented by int0 to int95, bit-packed using the 1024-bit interleaved layout. The first three rows represents the integers, and the last row represents the 1024 bit storage layout after bit-packing these integers. As can be seen, the first integer is placed in the first 32-bit word, while the second integer is placed in the next 32-bit word.

Furthermore, if W is not divisible by B, the last bit-packed integer of a word does not fit inside one word (called cross-case) and needs to be split across the current and next word. In our layout, the next word to w_i is w_i in the next 1024 bit instead of w_{i+1}, which is the case for non-interleaving case. Figure 3.2 shows an example of this where W is not divisible by B for W = 3 and B = 7. Each number inside the box shows the position of the integer inside the 1024 integers list. The splited bit-packed integers needs an OR instruction to be reconstructed.

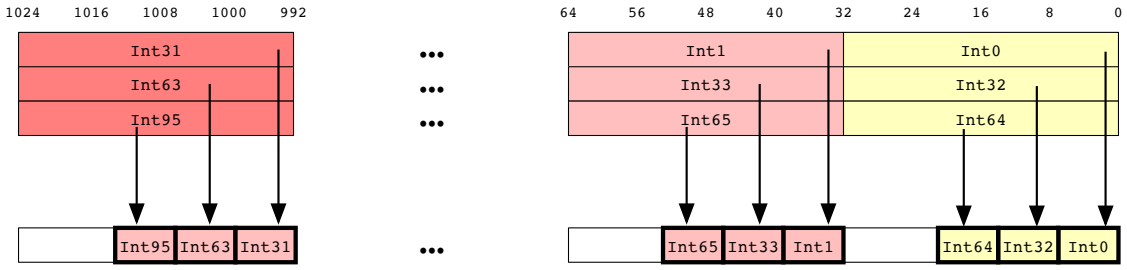


Figure 3.1: The 1024-bit interleaved data layout where W = 32 and B = 7.

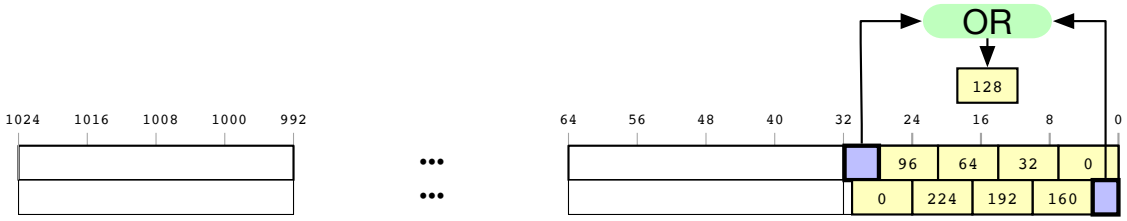


Figure 3.2: The 1024-bit interleaved data layout where W = 32 and B = 7.

3.1.1 Intermediate Word

If B is smaller than W, we can use a shorter word WP to bit-pack data, which leads to having more lanes and more parallelism during bit-unpacking. For example, if 3 bits are used to bit-pack a 32-bit integer, instead of storing it inside a 32-bit word, storing it inside an 8-bits word is possible. This change enables us to execute one SIMD instruction on four times more data. However, the resulting bit-unpacked data is of size WP and needs to be converted to W. To achieve this, a convert SIMD instruction is required, which is relatively expensive. Moreover, due to the shorter word size, it is more likely that the bit-packed data is stored in two words, which needs one additional OR instruction to be reconstructed.

3.2 Bit-unpacking

The bit-unpacking process implemented for our 1024-bit interleaved layout is provided as a library that consists of code for all variants of SIMD registers and scalar CPUs. At compile time, the best possible implementation, which is the implementation for the widest supported SIMD register on the targeted CPU, is auto-generated and compiled. For each

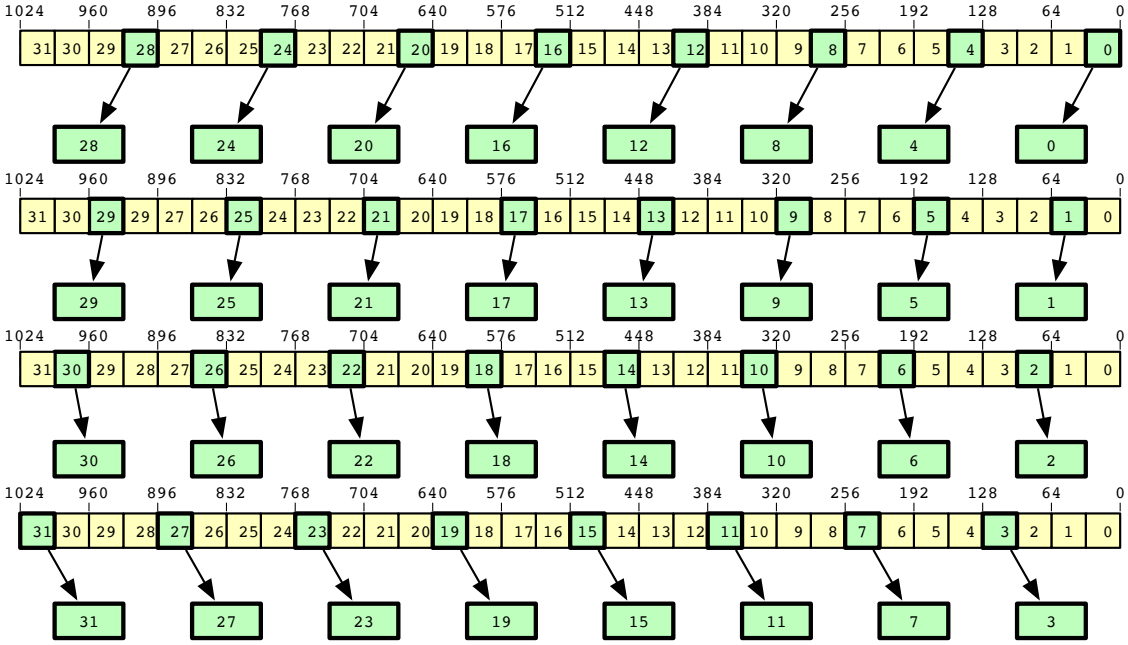


Figure 3.3: An example of the multi cursors technique applied to scalar bit-(un)packing.

possible combination of W and B, the library contains a hard-coded routine generated and stored in function tables indexed by B and W. Each routine processes 1024 values per iteration to ensure that results are always aligned to an integer that is multiple of 1024. In the following sections, we discuss the details of the optimizations to the scalar and vectorized implementations of our library.

3.2.1 Scalar Bit-Unpacking

Our storage layout is interleaved to exploit the SIMD capabilities in modern CPUs. However, this layout may introduce overhead for CPUs that lack SIMD registers. One possible optimization is to unroll the loop inside each function. However, this causes high register pressure (a measure for the availability of free registers) as for 8-bit, 16-bit, and 32-bit integers 128, 64, and 32 registers are needed. When the register pressure is high, the register contents must be spilled into memory and reloaded again, which negatively affects the performance. To solve this problem, the *multi cursor* technique can be used, where the input space is split into multiple distinct regions, and a single loop is used to iterate those regions in parallel by maintaining multiple cursors. However, to find the optimal number of distinct regions, all different cases need to be benchmarked. Figure 3.3 shows

an example of the multi cursor technique applied where $W = 32$ and the input space is divided into 8 regions. Therefore, instead of 32 registers, only 8 registers are needed.

Algorithm 1 shows the 1024-bit interleaved bit-unpacking scalar algorithm for different number of registers (R), W and B. In Line 9, the `Unpack` function is defined, which gets a variable *reg*, an *offset* to the beginning of the current bit-packed integer inside the variable, and B as input and returns the bit-unpacked integer as output. This function is implemented as a bit-wise AND operator applied on the variable shifted *offset* times to the right and a bit mask calculated by shifting the value 1, B times, and subtracting 1 from it. The algorithm's main body starts from Line 11, which needs to be repeated N times, where N is the number of regions defined for the multi cursor technique. Inside the main loop, all `if` and `for` loop blocks are unrolled in the actual implementation, which could not be shown in this algorithm due to space limitations. For each iteration, one register for each region is loaded with data from the memory. Then, for each register, the following steps are repeated W times.

- Based on the *offset* value, if the *offset* is equal to W, after calling the `UNPACK` function, the next word inside the memory needs to be loaded into the register. If the *offset* is bigger then W, it is a cross-case, and two values from different registers need to be combined with an OR operation. Otherwise, the `UNPACK` function needs to be called.
- The *offset* value needs to be changed to $(offset \ + B) \ \% \ W$.

An example of the generated implementation from this algorithm can be found in Listing 1, which shows a partial auto-generated unpacking routine that bit-unpacks 7-bit integers into 32-bit integers.

Algorithm 1: The 1024-bit interleaved bit-unpacking scalar algorithm.

```
1  in; // pointer to the input
2  out; // pointer to the output
3  R; // Number of registers
4  W; // Number of bits used to represent an integer in bit-unpacked
     form
5  B;  // Number of bits used to bit-pack an integer
6  L = 1024/W;
7  N = L/R;
8  offset = 0;
9  Function UNPACK(reg, offset, B):
10     return (reg >> offset) & ((1 << B) − 1);
11 for k ← 0 to N by 1 do
12     for i ← 0 to R by 1 do
13         reg_j ← ∗(in + i ∗ N + k);
14     for i ← 0 to W by 1 do
15         if offset == W then
16             in ← L + in;
17             for j ← 0 to R by 1 do
18                 reg_j ← ∗(in + j ∗ N + k);
19                 tmp ← UNPACK (reg_j, offset, B);
20                 ∗(out + k + j ∗ N + L ∗ i) ← tmp;
21             offset = offset+B ;
22         else if offset + B > W then
23             in ← L + in;
24             for j ← 0 to R by 1 do
25                 tmp ← UNPACK (reg_j,offset, W−offset);
26                 reg_j ← ∗(in + j ∗ N + k);
27                 tmp ← tmp | ( UNPACK (reg_j, 0, offset +B − W) << (W− offset));
28                 ∗(out + k + j ∗ N + L ∗ i) ← tmp;
29             offset←offset+B − W ;
30         else
31             for j ← 0 to R by 1 do
32                 tmp ← UNPACK (reg_j, offset, B);
33                 ∗(out + k + j ∗ N + L ∗ i) ← tmp;
34             offset = offset+B ;
```

3.2.2 Auto Vectorization

Modern compilers such as CLang and GCC are capable of *auto vectorizing* scalar code. Auto vectorization is the process of automatically transferring a scalar code to operate on a vector of values at once via SIMD instructions instead of operating on a single value at a time. If modern compilers are able to auto-vectorize the scalar implementation of our new layout, there is no need to explicitly implement vectorized bit unpacking using SIMD intrinsics. In the Section 3.3, we look into this question.

3.2.3 Vectorized Bit-Unpacking

Our 1024-bit interleaved layout makes it possible for all CPUs with SIMD capabilities to benefit from our layout. Algorithm 2 shows the 1024-bit interleaved bit-unpacking vectorized algorithm for different SIMD sizes. This algorithm works similarly to the scalar algorithm with two differences: all instructions are converted to SIMD instructions, and if W is not equal to WP during storing, the extract and convert instructions are used to transform the bit-unpacked integers to the desired form before storing.

```
static void unpack_bit_7_W_32_Wprime_32_R_8
(uint32_t *__restrict in, uint32_t *__restrict out)
  {
    uint32_t reg0, reg1, reg2, reg3, reg4, reg5, reg6, reg7, temp_reg;
    for (size_t i = 0; i < 4; ++i)
    {
      // Load 8 words into registers
      reg0 = * (in + 0 * 4  + i);
      reg1 = * (in + 1 * 4  + i);
      // ...
      // Extract and store first 7 bits of the loaded words as 32 bit integer
      temp_reg = (reg0 >> 0) & ((1 << 7) - 1);
      * (out + i + 0 * 4 + 32 * 0) = temp_reg;
      // ...
      // Extract and store second 7 bits of the loaded words as 32 bit integer
      temp_reg = (src0 >> 7) & ((1 << 7) - 1);
      * (out + i + 0 * 4 + 32 * 1) = temp_reg;
      //..
      // Extract and store third 7 bits of the loaded words as 32 bit integer
      temp_reg = (reg0 >> 14) & ((1 << 7) - 1);
      * (out + i + 0 * 4 + 32 * 2) = temp_reg;
      // ...
      // Extract and store third 7 bits of the loaded words as 32 bit integer
      temp_reg = (reg0 >> 21) & ((1 << 7) - 1);
      * (out + i + 0 * 4 + 32 * 3) = temp_reg;
      /// ...
      /// increase in by 32
      in += 32;
      // Extract and store forth 7 bits of the loaded words as 32 bit integer
      / /using or operation to concatenate the divided bit-packed int
      temp_reg = (reg0 >> 28) & ((1 << 4) - 1);
      reg0 = *  (in + 0  * 4 + i);
      temp_reg |= ((reg0 >> 0) & ((1 << 3) - 1)) << 4;
      *(out + i + 0 * 4 + 32 * 4) = temp_reg;
      // ...
      // repeats the above code 7 times
    }
  }
```

Listing 1: An example of scalar bit-unpacking implemented for B=7 and W=32.

Algorithm 2: The 1024-bit interleaved bit-unpacking SIMD algorithm

```
1  in; // pointer to the input out;// pointer to the output
2  R; W; WP; X // number of register bits
3  B;
4  l = 1024/X;
5  N = 1024/(X * R);
6  offset = 0;
7  Function UNPACK(reg, offset, B):
8      return AND_SIMD(rshift_SIMD(reg,offset), set1_SIMD((1UL << B) − 1)));
9  Function STORE(out, reg):
10     if WP == W then
11         store_SIMD(reg, out + k + j * N + l * i);
12     else
13         for m ← 0 to W/WP by 1 do
14             tmp ← convert_SIMD(extract_SIMD(reg, m));
15             store_SIMD(out + k + j * N + l * i + m, tmp);
16 for k ← 0 to N by 1 do
17     for i ← 0 to R by 1 do
18         reg_i ← load_SIMD(in + i * N + k);
19     for i ← 0 to WP by 1 do
20         if offset == W then
21             in ← l + in;
22             for j ← 0 to R by 1 do
23                 reg_j ← Load_SIMD(in + j * N + k);
24                 tmp ← UNPACK(reg_j, offset, B);
25                 STORE(out + k + j * N + l * i, tmp);
26             offset = offset + B ;
27         else if offset + B > W then
28             in ← l + in;
29             for j ← 0 to R by 1 do
30                 tmp ← UNPACK(reg_j, offset, W−offset);
31                 reg_j ← load_SIMD(in + j * N + k);
32                 tmp ← or_SIMD(tmp, lshift_SIMD( UNPACK (reg_j, 0, offset+B − W)
                    , (W − offset));
33                 STORE(out + k + j * N + l * i, tmp);
34             offset ← offset + B − W ;
35         else
36             for j ← 0 to R by 1 do
37                 tmp ← UNPACK(reg_j, offset, B);
38                 STORE(out + k + j * N + l * i, tmp);
39             offset = offset + B ;
```

3.3 Evaluation

In this section, we first evaluate the performance of the scalar implementation of our new bit-(un)packing layout and applied optimizations in terms of the performance. Then, the vectorized implementation of our new layout and the idea of using an intermediate word is benchmarked. Finally, the compiler's ability to automatically vectorize the scalar code is investigated. Note that all benchmarking in this section is done on two machines with different specifications shown in Table 3.1. Moreover, all results are show in plots where the X-axis denotes the number of bits used to bit-pack data, while the Y-axis shows the number of cycles spent per tuple.

	$M1$	$M2$
Operting System	macOS Catalina	Fedora 28
Processor Name	Quad-Core Intel Core i7	Intel(R) Core(TM) i9-7900X CPU
Processor Speed	2.9 GHz	3.30 GHz
Latest SIMD Support	avx2	avx512vl
L2 Cache (per Core)	256 KB	1024 KB

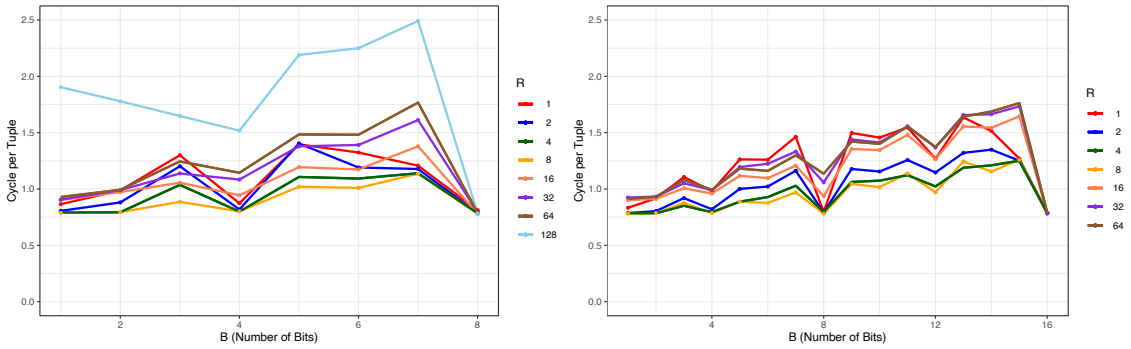
Table 3.1: Specifications of the machines used for benchmarking.

3.3.1 Scalar Unpacking

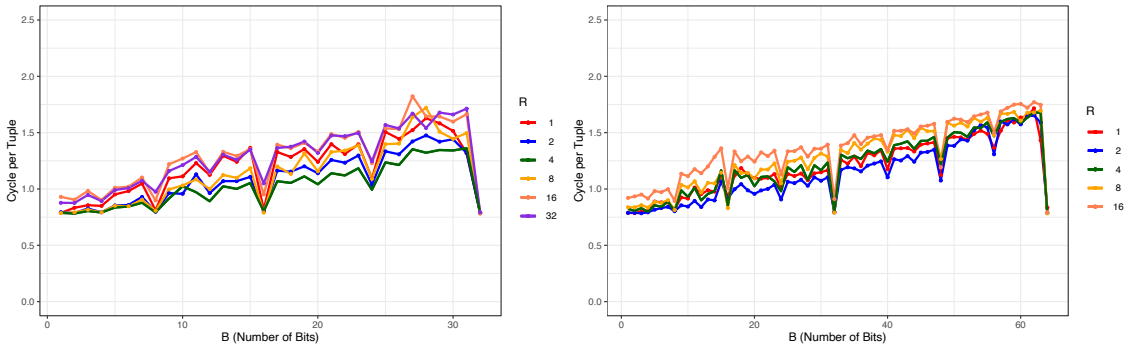
In this experiment, we aim to benchmark the performance of the scalar implementation of our new page layout to determine the optimal R value for each W. Note that our code is compiled using the `AppleClang 11.0.3` with the `-O3`, `-fno-slp-vectorize` and `-fno-vectorize` flags. The `-O3` flags instructs the compiler to optimize the code as much as possible. However, this flag also enables auto-vectorization, which is disabled using the `-fno-slp-vectorize` and `-fno-vectorize` flags.

Figure 3.4 shows the results of this experiment performed on $M1$. As can be seen, on M1, our multi-cursor technique is effective and the scalar implementation with 8 cursors for $W = 8$, 8 cursors for $W = 16$, 4 cursors for $W = 32$, and 2 cursors for $W = 64$ results in the best performance.

Moreover, we compared the performance of the scalar implementation of our layout to non-interleaved layout implementation for 32 bits integers. The result is shown in Figure 3.5. As can be seen, our 1024-bits interleaving layout does not introduce any overhead for scalar CPUs, and indeed with the help of the multi cursor technique, it can even achieve better performance.



(a) $W = 8$ **(b)** $W = 16$



(c) $W = 32$ **(d)** $W = 64$

Figure 3.4: Scalar bit-unpacking performance measured in terms of cycles per tuple for each B and W

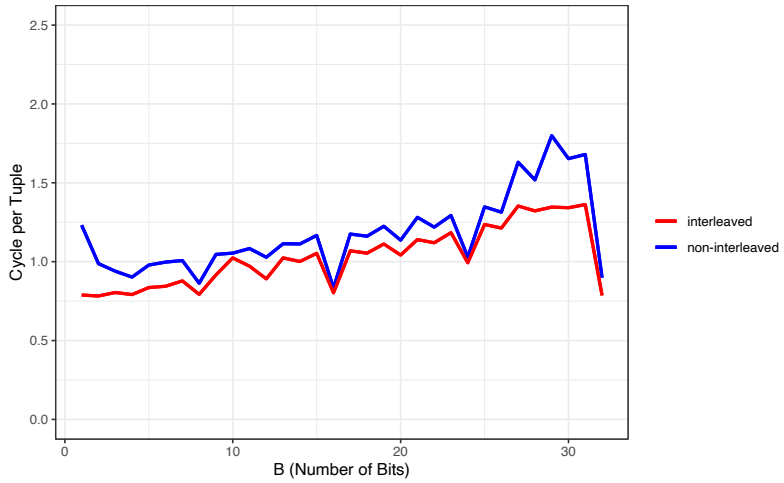


Figure 3.5: The performance of the non-interleaved layout vs interleaved layout in terms of cycles per tuple.

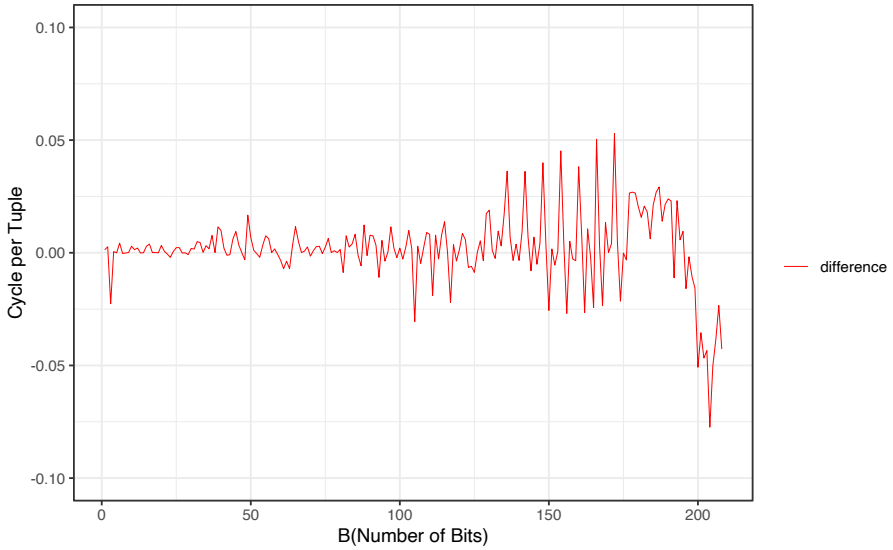


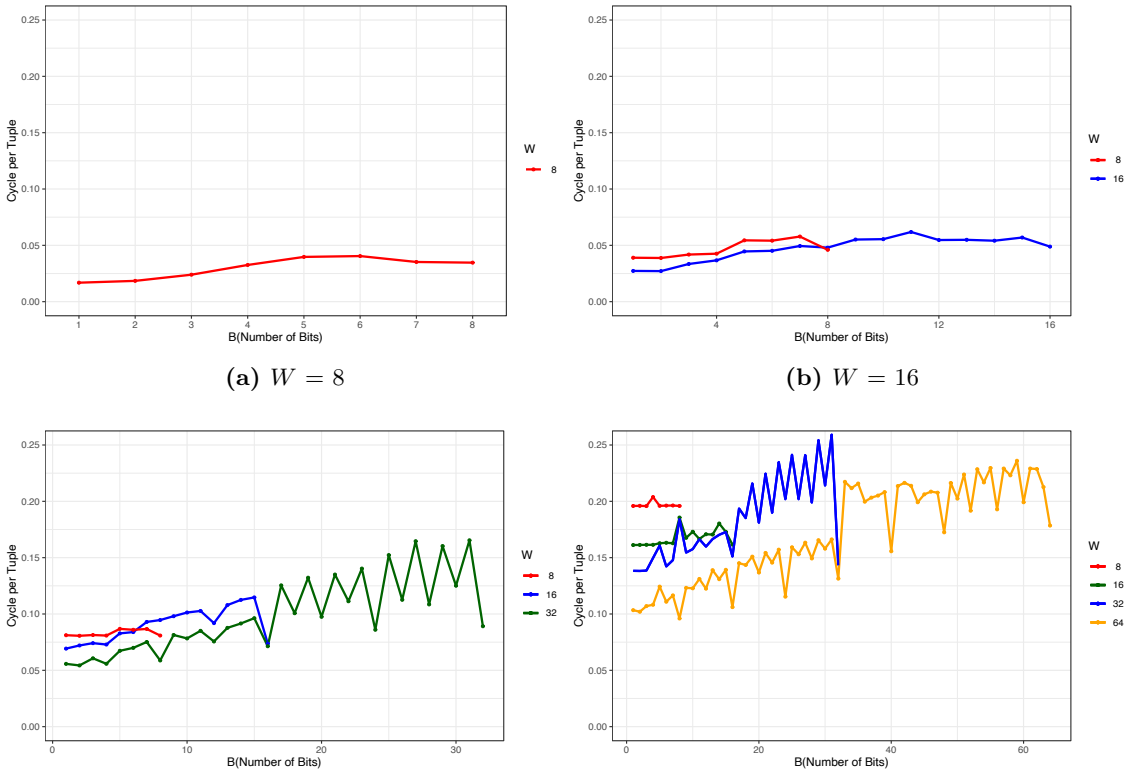
Figure 3.6: The performance of vectorized bit-unpacking in terms of cycles per tuple for each R (number of cursors).

3.3.2 Vectorized Unpacking and Intermediate Word

This experiment aims to 1) show how effective the multi cursor technique is on the vectorized implementation, 2) illustrate our vectorized implementation's performance, and determine whether using smaller intermediate word size (WP), as described in Section 3.1.1 is beneficial and 3) show the effect of using wider registers. Note that our code is compiled and executed using `Clang 4.2.1` with `-O3` and `-march=native` flags on M2.

Figure 3.6 shows the result of the first part of the experiment. Note that this plot is different from all other plots in this section, as X-axis shows all 216 possible functions generated from the permutation of all variables, R, W, and WP. Since M2 supports AVX-512 instructions, the R could have a value of 1 or 2. This plot shows the difference between all functions implemented with only one cursor, $R = 1$, versus two cursors, $R = 2$. As can be seen, on M2, our multi-cursor optimization technique is not effective and, indeed, for functions with $W = 64$ and $B > 32$ (the last part of the plot), it introduces an overhead of 0.1 cycles per tuple.

Figure 3.7 shows the results of the second part of this experiment performed on M2. As can be seen, on M2, using smaller intermediate word size is not beneficial for all smaller WPs. This is due to expensive SIMD convert instructions. However, it is possible that for the next generation of CPUs, the SIMD convert instruction becomes less expensive. Therefore, the idea of using smaller intermediate word size needs to be revisited in that



(a) $W = 8$ **(b)** $W = 16$

(c) $W = 32$ **(d)** $W = 64$

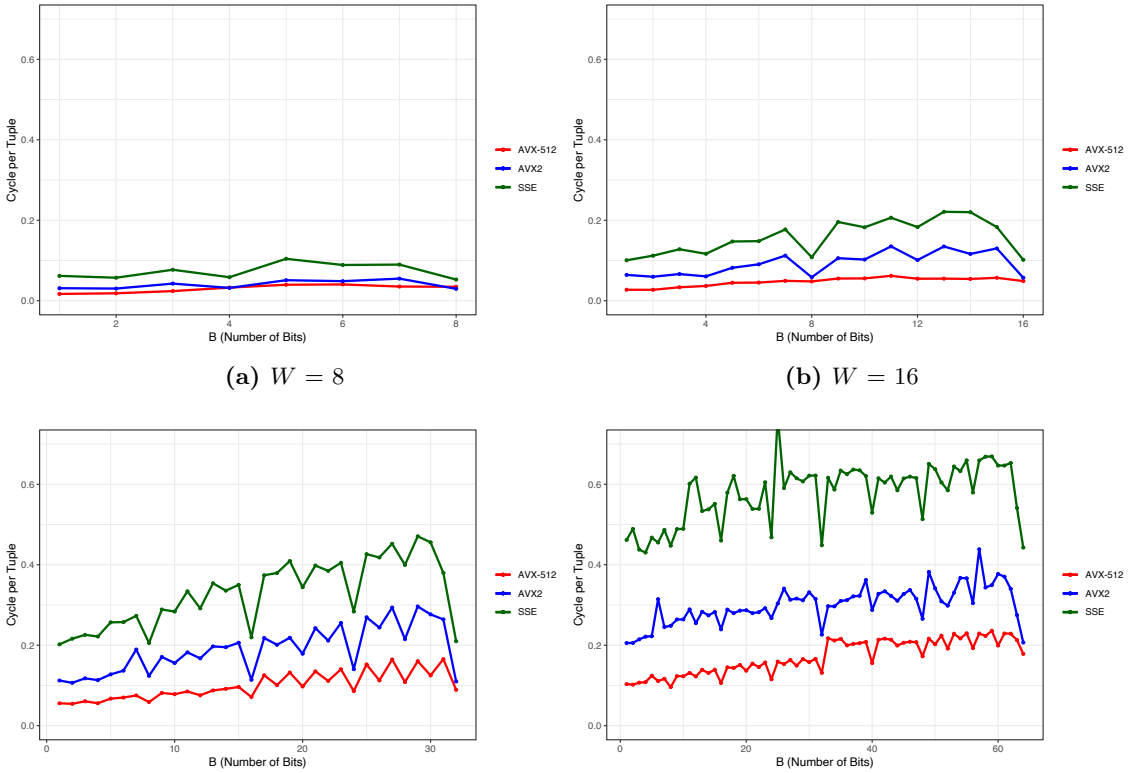
Figure 3.7: The performance of vectorized bit-unpacking in terms of cycles per tuple for each B and W.

case. Moreover, the results show that our new layout could be very efficient using SIMD instructions.

Figure 3.8 shows the results of the final part of this experiment performed on $M2$. As can be seen, on M2, the speed of bit-unpacking for our layout linearly increases with respect to a to the size of a SIMD register. Therefore, our layout achieves the best result for CPUs with different SIMD sizes. Moreover, we expect that our layout provides even 2x faster unpacking speed for the 1024 bit registers coming in the future (and 4x faster for 2048 bit registers).

3.3.3 Auto Vectorization

This experiment aims to 1) show which configuration for scalar code leads to better auto-vectorization and 2) illustrate to what extend the auto-vectorization by different compilers can simulate the result of explicit vectorized code. Note that our code is compiled and executed using `Clang 4.2.1` and `GCC 8` with `-O3` and `-march=native` flags on $M1$ and



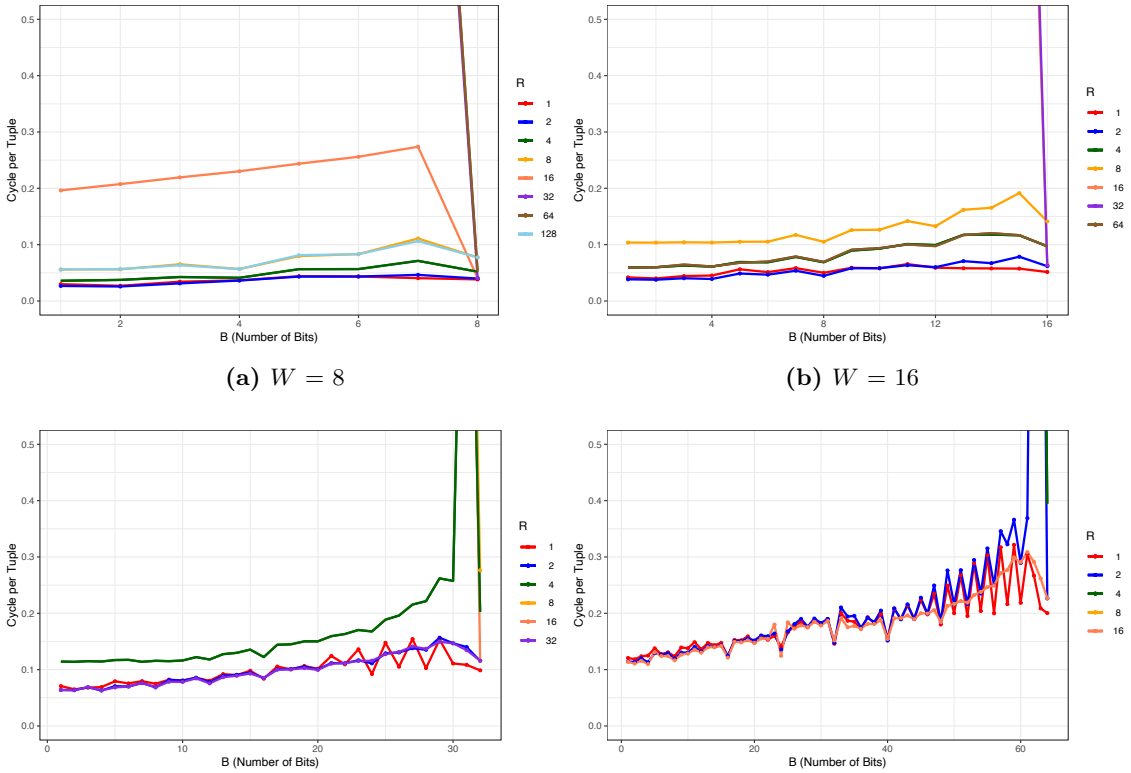
(a) $W = 8$ **(b)** $W = 16$

(c) $W = 32$ **(d)** $W = 64$

Figure 3.8: The performance of vectorized bit-unpacking in terms of cycles per tuple for each B and W.

$M2$.

Figure 3.9 shows the results of the first part of this experiment performed on $M2$. As can be seen, on $M2$, our multi-cursor optimization technique is not effective with auto-vectorizion, and scalar implementation with one cursor results in the best performance.

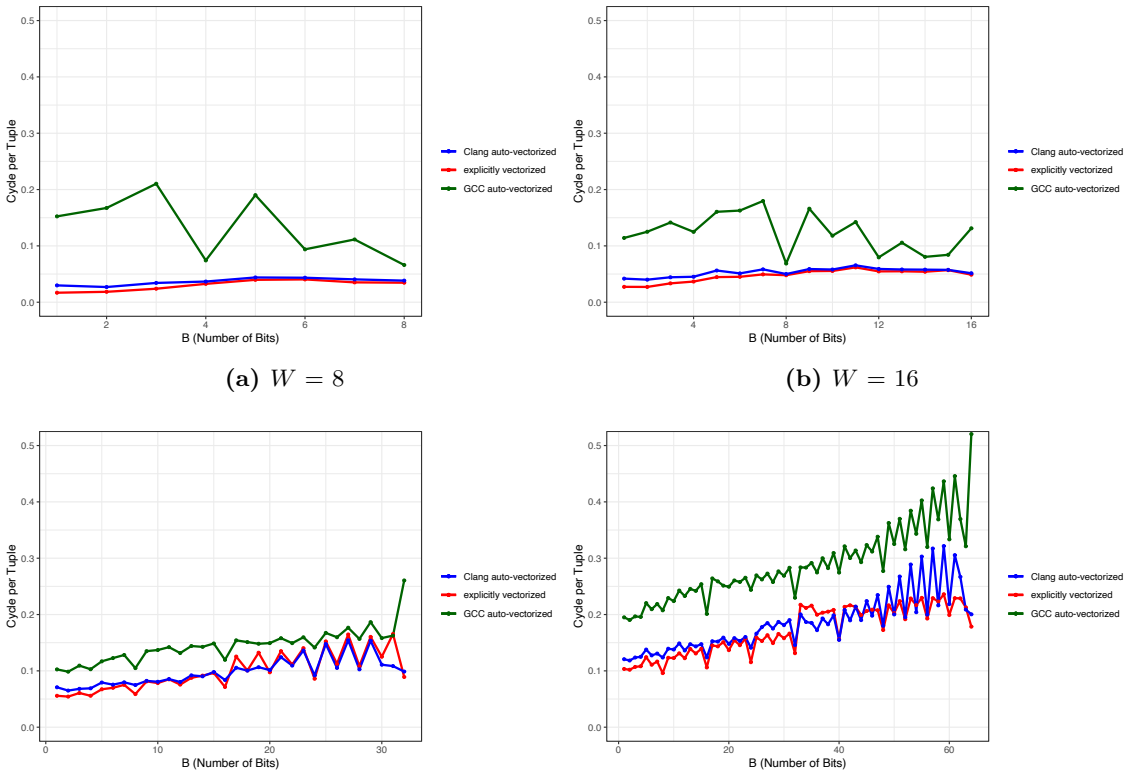


(a) $W = 8$ **(b)** $W = 16$

(c) $W = 32$ **(d)** $W = 64$

Figure 3.9: The performance of auto-vectorized bit-unpacking in terms of cycles per tuple for each possible B, W and R.

Figure 3.10 shows the results of second part of this experiment performed on $M2$. As can be seen, in most cases, explicitly vectorized code is two times faster, which is considerable. Figure 3.11 shows the results of second part of this experiment performed on $M1$. As can be seen, in cases where $W = 64$ and $B > 32$, the Clang compiler completely fails to auto vectorize the scalar code.



(a) $W = 8$ **(b)** $W = 16$

(c) $W = 32$ **(d)** $W = 64$

Figure 3.10: The performance of auto-vectorized bit-unpacking vs explicitly vectorized bit-unpacking in terms of cycles per tuple for each possible B, W.

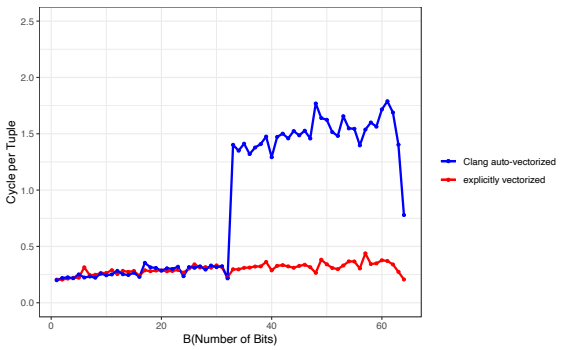


Figure 3.11: The performance of auto-vectorized bit-unpacking vs explicitly vectorized bit-unpacking in terms of cycles per tuple on M1, where $W = 64$;

4

Composable Functions

In this section, we define new composable functions, which are required to simulate black-box compression schemes. Furthermore, to optimize all these functions via SIMD instructions, we propose several novel algorithms. All these functions are implemented as a function that processes and returns 1024 values at a time. Note that 1024 is chosen for prototyping, and we are aiming to enlarge the vector size to 4096 (or even more).

4.1 Bit-Unpacking

The bit-unpack function takes a vector of 1024 bit-packed tuples and a bit-width value b, as input and returns a vector containing the bit-unpacked version of these inputs. For the bit-unpacking function, we use our new layout proposed in Section 3. To improve the compression ratio, instead of having a single bit-width for the entire compressed segment, we use a different bit-width for every 1024 tuples, which is stored in a mini-frame. However, this change might introduce latency to the system as it might be required to call a new bit-unpacking routine for every 1024 tuples.

4.1.1 Evaluation

To evaluate the latency issue described in the previous section, we designed an experiment where a sample of 2^{20} integers is extracted from all the 32-bits non-nullable integer columns of all datasets in the Public BI benchmark (3). Then, these samples are bit-packed in two scenarios, A and B. In scenario A, for each sample, the single best value b is determined, while in scenario B, for every 1024 tuples, the best value for b is determined. Note that the best b value is determined using the algorithm proposed by Heman (42), after subtracting

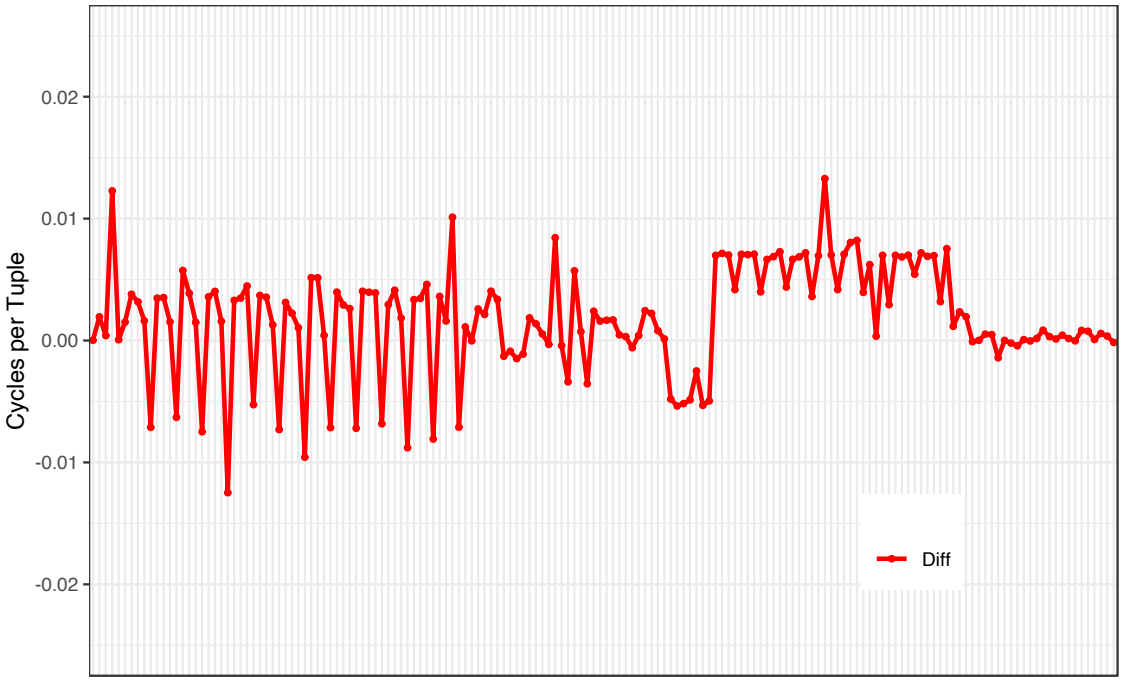


Figure 4.1: Difference between scenario A and B in terms of cycles per tuple.

the base value from all integers to make the experiment more realistic. Then, the bit-packed dataset is bit-unpacked once using a single determined b (Scenario A) and once a different b for every 1024 values (Scenario B). The difference is shown in Figure 4.1, where each point shows the difference between two scenarios for a given dataset. The detailed version of this figure it represented in Table 2. As can be seen in Figure 4.1, the difference is not significant, and indeed, in some cases, it is negative which shows the performance is improved. This is because the value b for each chunk might be smaller in scenario B, and as already shown, the smaller b results in faster bit-unpacking. To conclude, having different bit-widths makes our bit-unpacking flexible without additional latency, and in some cases, even leads to better performance. Note that the compression ratio will be benchmarked in Chapter 5 (composable PFOR) as the effect of bit-width is usually entangled with another variable and both needs to be benchmarked together.

4.2 Prefix Sum

The prefix-sum function takes an array of 1024 integers, $int_1, int_2, ..., int_{1023}$ and a base value, $base$, as input and computes the given input's prefix-sum as $(base + int_1), (base +$

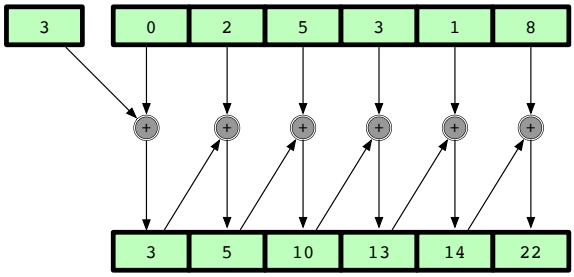


Figure 4.2: An example of a prefix-sum function.

int_1+int_2), ..., ($base$+int_1+...+int_{1023}). This layout is similar to D1. Figure 4.2 illustrates an example of this function. As can be seen, for each integer's prefix-sum computation, the prefix-sum of preceding integers needs to be known beforehand. This property of prefix-sum results in data dependency, and therefore, compilers fail to accelerate prefix sum via the loop unrolling technique.

As discussed in Section 2.5.2, the most efficient way to SIMDize prefix-sum is proposed by Zhang *et al.*, an extension of SIMD approach proposed by Lemire *et al.* to AVX-512 (35). However, in this approach, the data dependency still exists.

We propose a new transposed layout for tuples which minimizes the prefix-sum's data dependency. Our new layout changes the order of tuples, so there is no data dependency between SIMD lanes. This change enables us to compute the prefix-sum using only one SIMD ADD instruction instead of multiple ADD and SHIFT instructions in Zhang *et al.*'s approach. Moreover, as long as all operators are aware of the new order of tuples, there is no need to recover the original placement of tuples. However, if it is not the case, the transpose operator needs to be applied to tuples. Furthermore, our new prefix-sum operator could be combined with our transpose operator, resulting in a prefix-sum operator that keeps the original order. Our new transposed layout and optimized transpose algorithm are discussed in the following sections.

4.2.1 Transposed Layout

In our transposed storage layout, every 1024 W-bits integers are distributed among L vertical groups of $1024/L$ integers, $L_1, L_1, ..., L_W$, where L is the number of W-bit wide SIMD lanes a 1024 bit register has, and W is 8, 16, 32, and 64, the number of bits used to represent an integer. Our transposed layout distributes the integers among L groups in the following order: the ith integer goes to the $\lfloor \frac{i-32}{32} \rfloor$th vertical group, i.e., the 0th

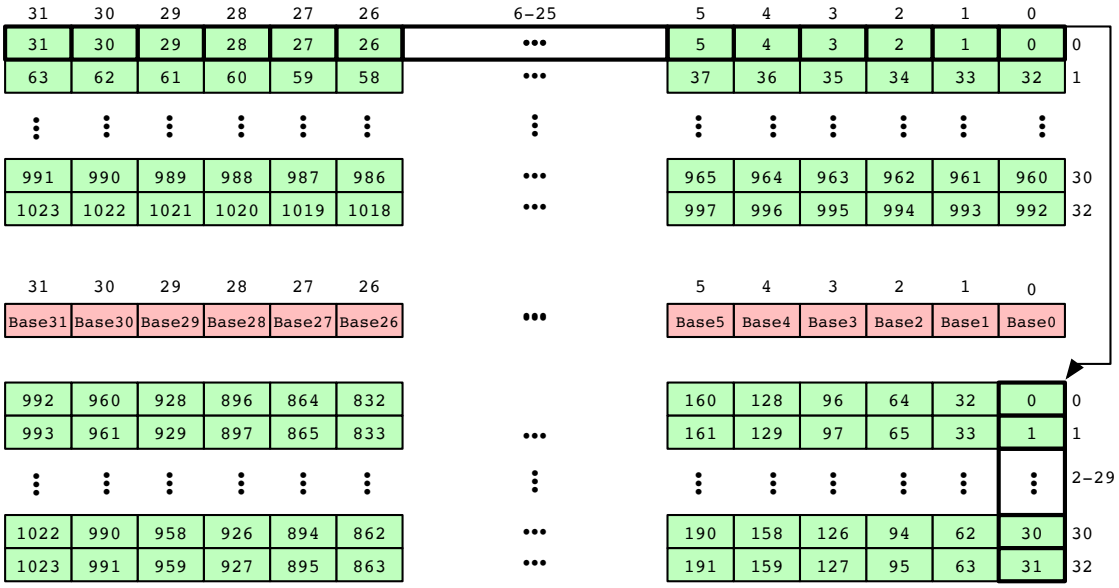


Figure 4.3: The transposed layout for the prefix-sum calculation.

integer goes to the 0th group, the 1st integer goes to 0th group and so on until $L-1$. After that, the Lth integer goes to the 1st group, and so on.

Our new transposed layout removes the data dependency for integers inside a SIMD lane. However, instead of having one base per 1024 integers, each SIMD lane needs to have a base value, which results in extra (L - 1) W-bit integers (1024 bit in total). Figure 4.3 shows an example of 32-bits integers which are transformed into our new layout.

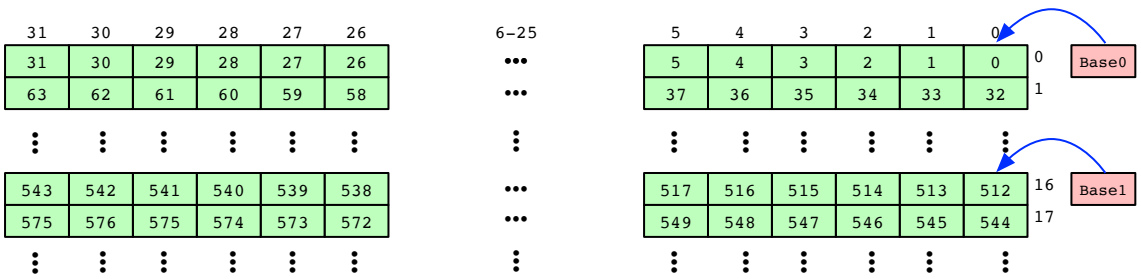


Figure 4.4: The 2-cursors layout for prefix-sum calculation

4.2.2 Evaluation

To evaluate the performance of our proposed layout, we implemented and benchmarked the following variations of prefix-sum. Note that all these variations are implemented as a function that calculates the prefix-sum of 1024 32-bit integers.

- Prefix-Sum: the implementation of prefix-sum for our layout using the AVX-512 instructions set shown in Listin 2. In this implementation, 65 SIMD LOAD, 64 SIMD ADD, and 64 STORE instructions are used.

```
1   void prefix_sum(uint32_t * __restrict__ in, uint32_t  * __restrict__ out
2   , uint32_t *__restrict__ base) {
3       __m512i *_in = reinterpret_cast<__m512i *>(in);
4       __m512i *_out = reinterpret_cast<__m512i *>(out);
5       __m512i base1 = _mm512_loadu_si512(base + 0);
6       __m512i base2 = _mm512_loadu_si512(base + 16);
7       __m512i tmp1 = _mm512_loadu_si512(_in + 2 * 0);
8       __m512i tmp2 = _mm512_loadu_si512(_in + 2 * 0 + 1);
9       tmp1 = _mm512_add_epi32(base1, tmp1);
10      tmp2 = _mm512_add_epi32(base2, tmp1);
11      _mm512_storeu_si512(_out, tmp1);
12      _mm512_storeu_si512(_out + 1, tmp1);
13      for (int i = 0; i < 32; ++i) {
14          tmp1 = _mm512_add_epi32(_mm512_loadu_si512(_in + 2 * i), tmp1);
15          _mm512_storeu_si512(_out + 2 * i, tmp1);
16          tmp2 = _mm512_add_epi32(_mm512_loadu_si512(_in + 2 * i + 1), tmp2);
17          _mm512_storeu_si512(_out + 2 * i + 1, tmp2);
18      }
19  }
```

Listing 2: The implementation of prefix_sum function for `32-bit int`.

- Prefix-Sum_and_New-Transposed: the implementation of prefix-sum for our layout combined with the transposed operator to keep the order of tuples.
- D1-SSE: the implementation of D1 using SSE registers shown in Listing 3, proposed by Lemire *et al.* (35).
- D1-AVX2: the implementation of D1 using AVX2 registers shown in Listing 4.
- D1-AVX-512: the implementation of D1 using AVX-512 registers proposed by Zhang *et al.* (53) shown in Listing 5.
- Prefix-SUM_and_Scatter-Transposed: the implementation of prefix-sum for our layout combined with the transposed operator implemented using the scatter instruction (explained in Section 4.2.1) shown in Listing 6.

```
1   void d1_sse(uint32_t *__restrict__ in , uint32_t *__restrict__ out,
2   uint32_t base) {
3       __m128i tmp;
4       __m128i prev;
5       __m128i *_in = reinterpret_cast<__m128i *>(in);
6       __m128i *_out = reinterpret_cast<__m128i *>(out);
7       // i = 0;
8       tmp = _mm_loadu_si128(_in);
9       tmp = _mm_add_epi32(tmp, _mm_slli_si128(tmp, 8));
10      tmp = _mm_add_epi32(tmp, _mm_slli_si128(tmp, 4));
11      prev = _mm_add_epi32(tmp, _mm_set1_epi32(base));
12      _mm_storeu_si128(_out, prev);
13      for (int i = 1; i < 256; ++i) {
14          tmp = _mm_loadu_si128(_in + i);
15          tmp = _mm_add_epi32(tmp, _mm_slli_si128(tmp, 8));
16          tmp = _mm_add_epi32(tmp, _mm_slli_si128(tmp, 4));
17          prev = _mm_add_epi32(tmp, _mm_shuffle_epi32(prev, 0xff));
18          _mm_storeu_si128(_out + i, prev);
19      }
20  }
```

Listing 3: The implementation of d1_sse function for `32-bit int`.

- 2-cursors_Scalar: the implementation of prefix-sum using multi-cursor techniques with 2 cursors shown in Listing 7. The scalar implementation of prefix-sum could be enhanced with multi-cursor techniques. However, multiple bases need to be added to the layout instead of one. Figure 4.4 shows an example of multi cursor techniques applied to the D1 layout.
- 4-cursor_Scalar: the implementation of prefix-sum using multi-cursor techniques with 4 cursors shown in Listing 8.
- D1-Scalar-Clang: the implementation of prefix-sum using scalar code compiled by the Clang compiler. The generated assembly code shows that the loop inside the function is unrolled 3 times.
- D1-Scalar-GCC: the implementation of prefix-sum using scalar code compiled by GCC, which does not use unrolling.

Figure 4.5 shows the result of the experiment. The Y-axis indicates the number of cycles per tuple, while the X-axis shows its implementation. As can be seen, our layout is nearly 10x faster than the fastest possible implementation of the prefix sum, D1-AVX-512. If the end user wants tuples in the right order, the combination of our transposed layout with the transposed function is still 2x faster than D1-AVX-512. Moreover, the comparison

```
void d1_avx2(uint32_t *__restrict__ in, uint32_t *__restrict__ out,
uint32_t base) {
    __m256i tmp;
    __m256i prev;
    __m256i mask;
    __m256i *_in = reinterpret_cast<__m256i *>(in);
    __m256i *_out = reinterpret_cast<__m256i *>(out);
    // i = 0;
    tmp = _mm256_loadu_si256(_in);
    mask = _mm256_permute2x128_si256(tmp, tmp, _MM_SHUFFLE(0, 0, 3, 0));
    tmp = _mm256_add_epi32(tmp, _mm256_alignr_epi8(tmp, mask, 16 - 16));
    mask = _mm256_permute2x128_si256(tmp, tmp, _MM_SHUFFLE(0, 0, 3, 0));
    tmp = _mm256_add_epi32(tmp, _mm256_alignr_epi8(tmp, mask, 16 - 8));
    mask = _mm256_permute2x128_si256(tmp, tmp, _MM_SHUFFLE(0, 0, 3, 0));
    tmp = _mm256_add_epi32(tmp, _mm256_alignr_epi8(tmp, mask, 16 - 4));
    prev = _mm256_add_epi32(tmp, _mm256_set1_epi32(base));
    _mm256_storeu_si256(_out, prev);
    for (int i = 1; i < 128; ++i) {
        tmp = _mm256_loadu_si256(_in + i);
        mask = _mm256_permute2x128_si256(tmp, tmp, _MM_SHUFFLE(0, 0, 3, 0));
        tmp = _mm256_add_epi32(tmp, _mm256_alignr_epi8(tmp, mask, 16 - 16));
        mask = _mm256_permute2x128_si256(tmp, tmp, _MM_SHUFFLE(0, 0, 3, 0));
        tmp = _mm256_add_epi32(tmp, _mm256_alignr_epi8(tmp, mask, 16 - 8));
        mask = _mm256_permute2x128_si256(tmp, tmp, _MM_SHUFFLE(0, 0, 3, 0));
        tmp = _mm256_add_epi32(tmp, _mm256_alignr_epi8(tmp, mask, 16 - 4));
        prev = _mm256_add_epi32(tmp, _mm256_set1_epi32(_mm256_extract_epi32(prev, 7)));
        _mm256_storeu_si256(_out + i, prev);
    }
}
```

Listing 4: The implementation of $\mathrm{d1}_{avx2}$ *function for* 32-bit `int`.

of the performance of D1 implemented by SEE, AVX2, and AVX-512 shows that the performance of the proposed algorithm by Lemire *et al.* and Zhang *et al.* are not linearly increasing. This indicates that these algorithms cannot take complete advantage of new wider registers. This is different in our proposed layout, where the number of instructions does not increase for wider registers.

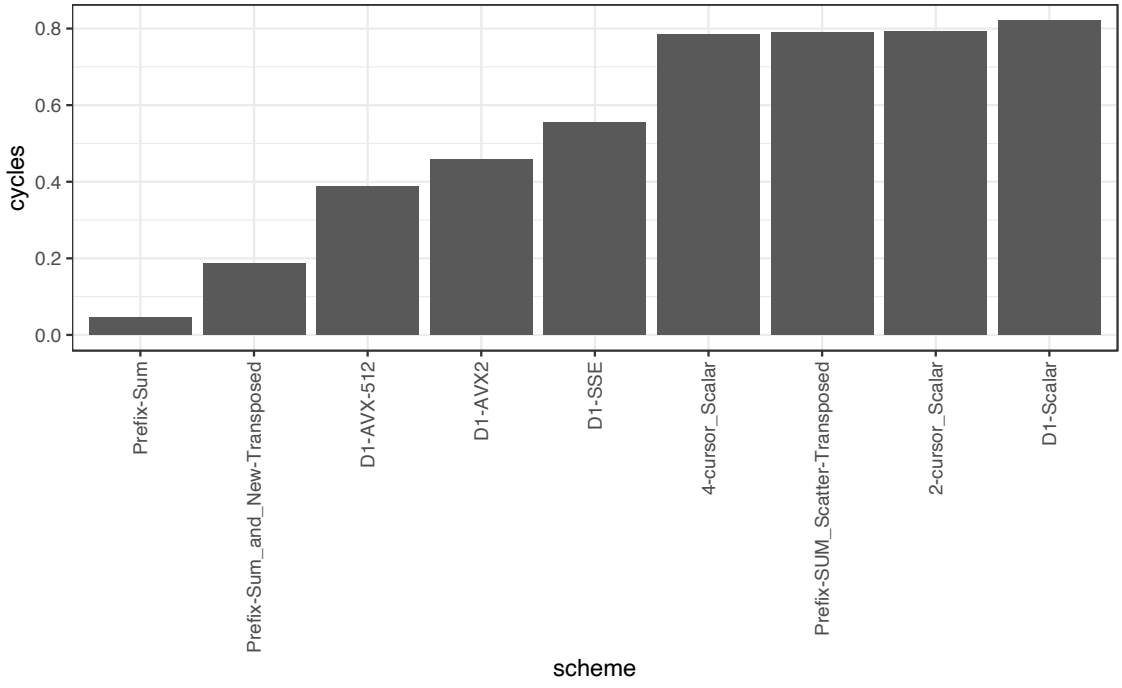


Figure 4.5: The performance of different implementation of prefix-sum in terms of cycles per tuple.

```
__m512i _mm512_slli_si512(__m512i x, int k) {
    __m512i ZERO = _mm512_setzero_si512();
}

__m512i PrefixSum(__m512i x) {
    __m512i ZERO = _mm512_setzero_si512();
    x = _mm512_add_epi32(x, _mm512_alignr_epi32(x, ZERO, 16 - 1));
    x = _mm512_add_epi32(x, _mm512_alignr_epi32(x, ZERO, 16 - 2));
    x = _mm512_add_epi32(x, _mm512_alignr_epi32(x, ZERO, 16 - 4));
    x = _mm512_add_epi32(x, _mm512_alignr_epi32(x, ZERO, 16 - 8));
    return x; // local prefix sums
}

void d1_avx_512(uint32_t * __restrict__ in, uint32_t * __restrict__ out,
uint32_t base) {
    __m512i tmp;
    __m512i prev;
    __m512i *_in = reinterpret_cast<__m512i *>(in);
    __m512i *_out = reinterpret_cast<__m512i *>(out);
    // i = 0;
    tmp = _mm512_loadu_si512(_in);
    prev = _mm512_add_epi32(PrefixSum(tmp), _mm512_set1_epi32(base));
    _mm512_storeu_si512(_out, prev);
    for (int i = 1; i < 64; ++i) {
        tmp = _mm512_loadu_si512(_in + i);
        prev = _mm512_add_epi32(PrefixSum(tmp), _mm512_set1_epi32(*(out + i * 16 - 1)));
        _mm512_storeu_si512(_out + i, prev);
    }
}
```

Listing 5: The implementation of D1_AVX-512 function for `32-bit int`.

```
void prefix_sum_and_scatter(uint32_t * __restrict__ in, uint32_t * __restrict__ out,
uint32_t * __restrict__ base) {
__m512i vindex1 = _mm512_set_epi32(480, 448, 416, 384, 352, 320, 288,
256, 224, 192, 160, 128, 96, 64, 32, 0);
__m512i *_in = reinterpret_cast<__m512i *>(in);
__m512i *base_ = reinterpret_cast<__m512i *>(base);
__m512i tmp1;
__m512i tmp2;
__m512i base1 = _mm512_loadu_si512(base_ + 0);
__m512i base2 = _mm512_loadu_si512(base_ + 1);
tmp1 = _mm512_loadu_si512(_in);
tmp1 = _mm512_add_epi32(tmp1, base1);
tmp2 = _mm512_loadu_si512(_in + 1);
tmp2 = _mm512_add_epi32(tmp2, base2);
_mm512_i32scatter_epi32(out, vindex1, tmp1, 4);
_mm512_i32scatter_epi32(out + 512, vindex1, tmp2, 4);
for (int i = 1; i < 32; ++i) {
    tmp1 = _mm512_add_epi32(tmp1, _mm512_loadu_si512(_in + (2 * i) + 0));
    tmp2 = _mm512_add_epi32(tmp2, _mm512_loadu_si512(_in + (2 * i) + 1));
    _mm512_i32scatter_epi32(out + i, vindex1, tmp1, 4);
    _mm512_i32scatter_epi32(out + i + 512, vindex1, tmp2, 4);
}
}
```

Listing 6: The implementation of Prefix-SUM_and_Scatter-Transposed function for 32-bit int.

```
void scalar_2cursor(uint32_t * __restrict__ in, uint32_t * __restrict__ out,
uint32_t * __restrict__ base) {
    out[0] = in[0] + base[0];
    out[512] = in[512] + base[2];
    for (int i = 1; i < 512; i++) {
        out[i] = out[i - 1] + in[i];
        out[i + 512] = out[i + 512 - 1] + in[i + 512];
    }
}
```

Listing 7: The implementation of 2-cursors_Scalar function for 32-bit int.

```
1       void
2     scalar_4cursor(uint32_t *__restrict__ in, uint32_t *__restrict__ out,
3     uint32_t *__restrict__ base) {
4         out[0] = in[0] + base[0];
5         out[256] = in[256] + base[1];
6         out[512] = in[512] + base[2];
7         out[768] = in[768] + base[3];
8         for (int i = 1; i < 256; i++) {
9             out[i] = out[i - 1] + in[i];
10            out[i + 256] = out[i + 256 - 1] + in[i + 256];
11            out[i + 512] = out[i + 512 - 1] + in[i + 512];
12            out[i + 768] = out[i + 768 - 1] + in[i + 768];
13        }
14    }
```

Listing 8: The implementation of 4-cursor_Scalar function for `32-bit int`.

4.3 Plus

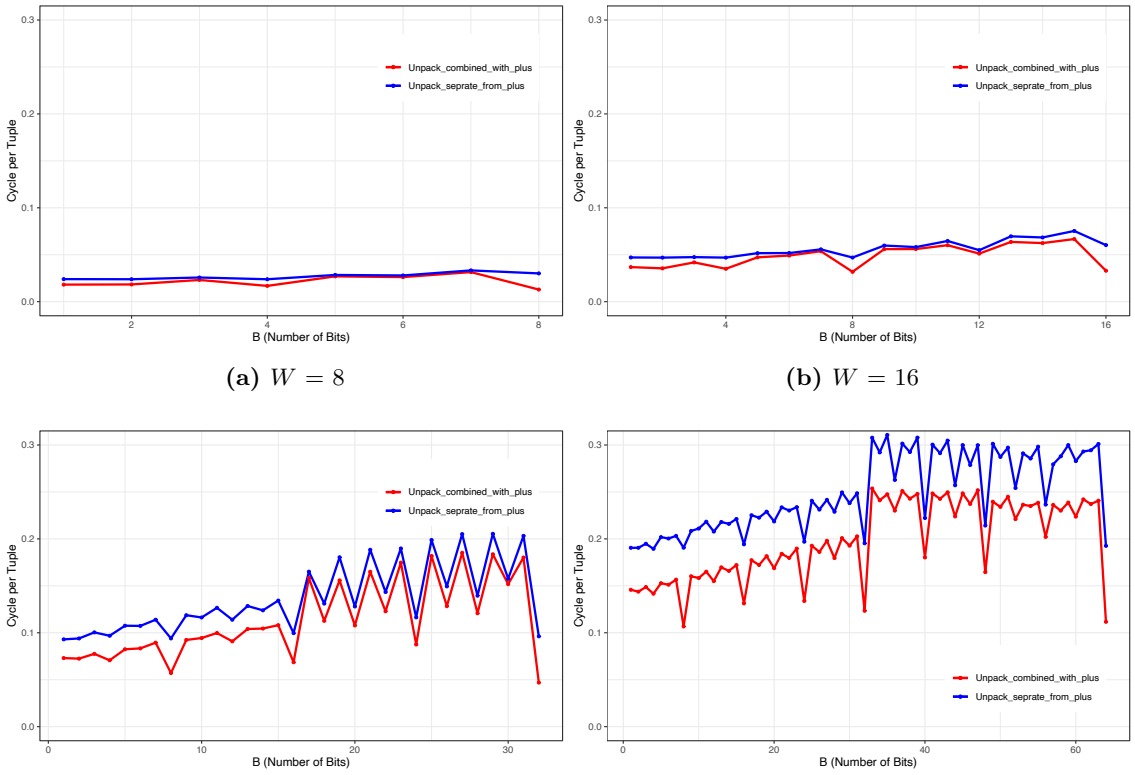
The plus function takes a vector of 1024 integers, int_1, int_2, ..., int_{1023} and a base value, *base*, as input and adds the base value to all integers. The base value is stored inside a mini-frame, as explained in Section 1.3. The plus function is usually needed for the FOR-like compression algorithms to add the reference to compressed integers. However, in whitebox compression model, the plus function leads to executing additional instructions to load data from memory and store them back, which could be avoided if the plus function was combined with its previous function in the evaluation tree, which is always bit-unpacking. Moreover, the combination of the bit-unpacking and plus functions guarantees that the SIMD ADD instruction is used if there is a possibility.

4.3.1 Intermediate Base

As discussed in Section 3.1.1, the idea of using the intermediate word to bit-unpack data is not beneficial as the SIMD CONVERT instruction is expensive. However, after combining bit-unpacking with the plus function, this idea is worth being benchmarked again, as we can potentially use more lanes, and therefore have more parallelism, to bit-unpack and add the base value to inputs. Therefore, if the base value could be represented with a shorter word of size WP ($WP < W$) more parallelism can be used during the bit-unpacking and plus operations.

4.3.2 Evaluation

In this experiment, we benchmarked the performance of the plus function in two cases: when it is combined with bin-unpacking and when it is separate. Figure 4.6 shows the results of this experiment performed on M2. As can be seen, the combined version results in better performance. Moreover, the best possible performance of bit-unpacking using an intermediate word shown in Figure 3.7 is even worse than the case where plus and bit-unpacking are combined without using an intermediate word. This shows that using smaller intermediate words is also not beneficial for the combination case. This is due to expensive SIMD CONVERT instructions. However, it is possible that for the next generation of CPUs, the SIMD CONVERT instruction becomes less costly. Therefore, using smaller intermediate word size needs to be revisited again for this case. Finally, as the performance of bit-unpacking is very close to bit-unpacking combined with plus, the bit-unpacking library could be replaced with the library of the combination of plus and bit-unpacking. If for other whitebox compression schemes, only bit-unpacking is needed,



(a) $W = 8$ **(b)** $W = 16$

(c) $W = 32$ **(d)** $W = 64$

Figure 4.6: The performance of the plus function combined with bit-unpacking vs the separate version in terms of cycles per tuple.

the base parameter could be set to 0. This change significantly reduces the compile time, while the performance remains almost the same.

4.4 Transpose

The transpose function takes a vector of 1024 tuples with the transposed layout described in Section 4.2.1 as input and restores the original layout. Listing 9 shows the implementation of a simple transpose function, called the scalar implementation, as no SIMD instruction is used. As can be seen, inside the inner loop, each element in the input vector is placed into its original position in the output vector.

However, this implementation might be accelerated using the new SCATTER instructions introduced for AVX-512. Listing 10 shows the implementation of the transpose function using SCATTER instructions for 32-bit unsigned integers, called the SIMD implementation. In this implementation, a temporary vector is generated, holding the po-

```
void scalar_transpose(const ANY *__restrict__ in, ANY *__restrict__ out) {
    for (i = 0; i < 32; ++i) {
        for (j = 0; j < 32; ++j) {
            out[32 * j + i] = in[32 * i + j];
        }
    }
}
```

Listing 9: The scalar implementation of the transpose function.

```
void SIMD_transpose(uint32_t *__restrict__ in,
uint32_t *__restrict__ out) {
    __m512i vindex1 = _mm512_set_epi32(480, 448, 416, 384, 352, 320, 288,
    256, 224, 192, 160, 128, 96, 64, 32, 0);
    __m512i *_in = reinterpret_cast<__m512i *>(in);
    for (i = 0; i < 32; ++i) {
        _mm512_i32scatter_epi32(out + i, vindex1,
        _mm512_loadu_si512(_in + (2 * i) + 0), 4);
        _mm512_i32scatter_epi32(out + i + 512, vindex1,
        _mm512_loadu_si512(_in + (2 * i) + 1), 4);
    }
}
```

Listing 10: The SIMD implementation of transpose function for `uint32_t`.

sitions where the input elements should be copied to. In each iteration, 32 elements are copied into their correct position in the output vector using the SCATTER and LOAD instructions.

However, the SCATTER instruction is costly and could have a delay of 12 cycles. To solve this problem, we propose a new algorithm that uses less expensive SIMD instructions such as UNPACKLO, UNPACKHI, SHUFFLE, and PERMUTE. This algorithm uses 17 SIMD registers, which is less than the total number of registers in a CPU. Thus, this algorithm does not incur register pressure.

To explain our SIMD transpose algorithm, as an example, we consider a matrix of 32×32 elements and a CPU that supports AVX-512 instructions. 1024 elements of the vector input can be represented as a matrix of 32×32 elements. We divide the matrix into four matrices of 16×16, where the same steps are applied to all four matrices. The steps of the algorithm are as follows:

- Load all the tuples into 16 registers $r_0,\ r_1, ...,\ r_{15}$, as shown in Figure 4.7.
- Repeat the following steps 5 times where s initially set to 1, shows the number of tuples that are together during interleaving and i is initially set to 0.

- Starting from r_0, group all the registers into 8 groups of r_n and r_m where $n-m=i$. For example for $i=0$, r_{14} is grouped with r_{15} , as shown in Figure 4.8;
- For each group, interleave the first half of two registers and store the result in a temporary register. Then, interleave the second half of two registers and store the result into r_j where $j > i$. After that copy the temporary register into r_i. Note that the interleaving is done based on the value of s. For example, if $s=2$, every two tuples are paired with each other. Thus, after interleaving, every two tuples of one register are followed by two tuples from the other register.
- Set i to $2*i+1$
- Set s to $s+1$

- Store each register at the right position. Figure 4.12 shows the content of registers before the store. As can be seen, each register needs to be stored in random order.

Note that Figures 4.9, 4.10, and 4.11 show the content of registers before steps 2, 3, and 4, respectively. Also, it is worth mentioning that our proposed algorithm can be extended to a matrix with any size and a CPU with any SIMD register size. Considering a matrix of size $N \times M$ and a SIMD register of size K, all matrices of size $K \times K$ inside the main matrix could be transposed using the above-mentioned algorithm. Then, all remaining areas should be transposed in a scalar way.

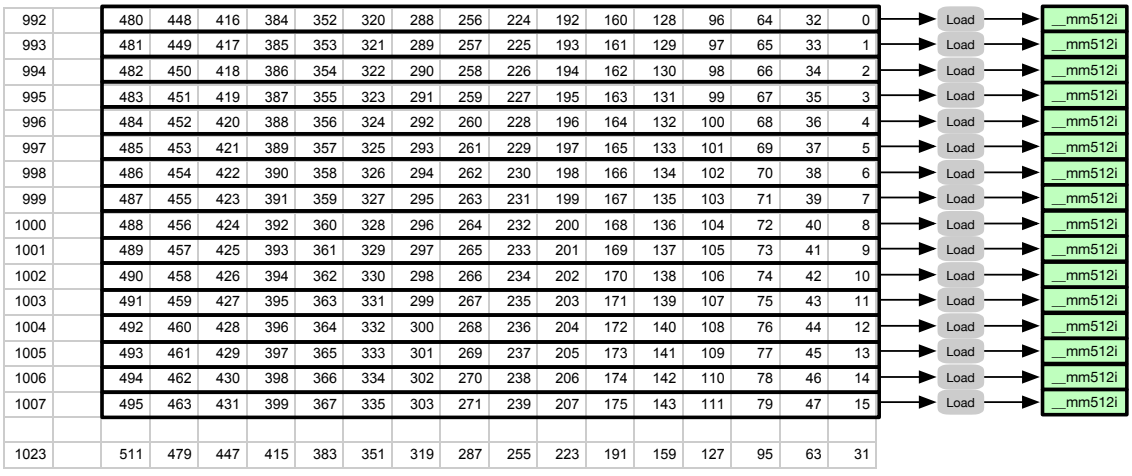


Figure 4.7: Our new transpose algorithm, the loading step.

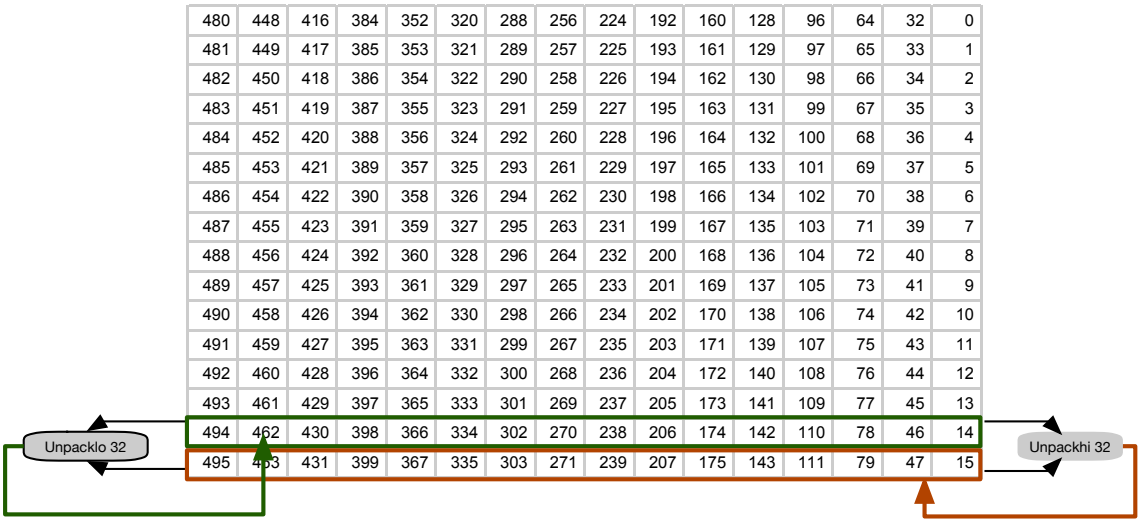


Figure 4.8: Our new transpose algorithm where $i = 0$.

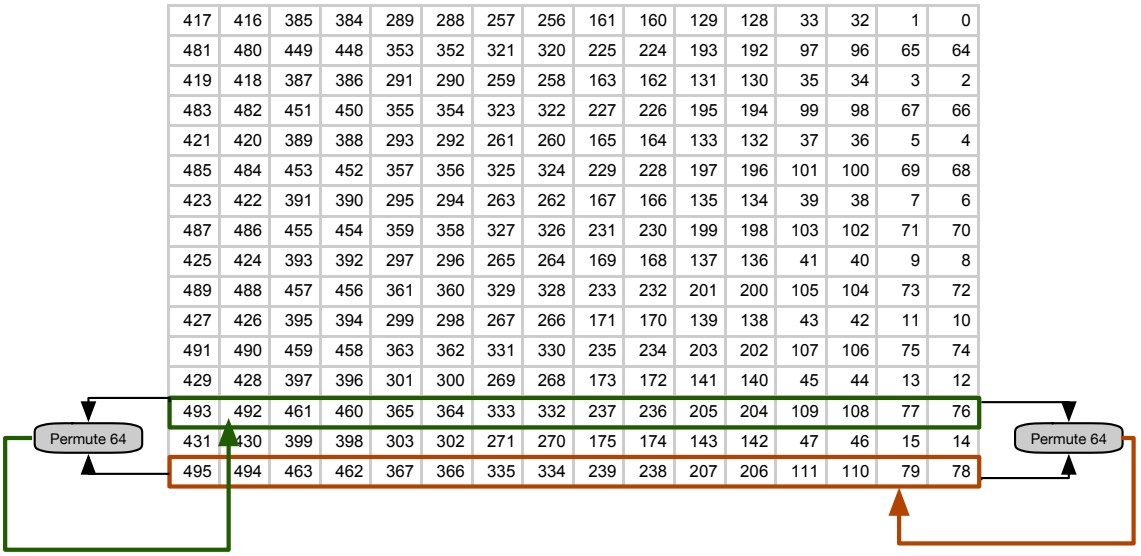


Figure 4.9: Our new transpose algorithm where $i = 1$.

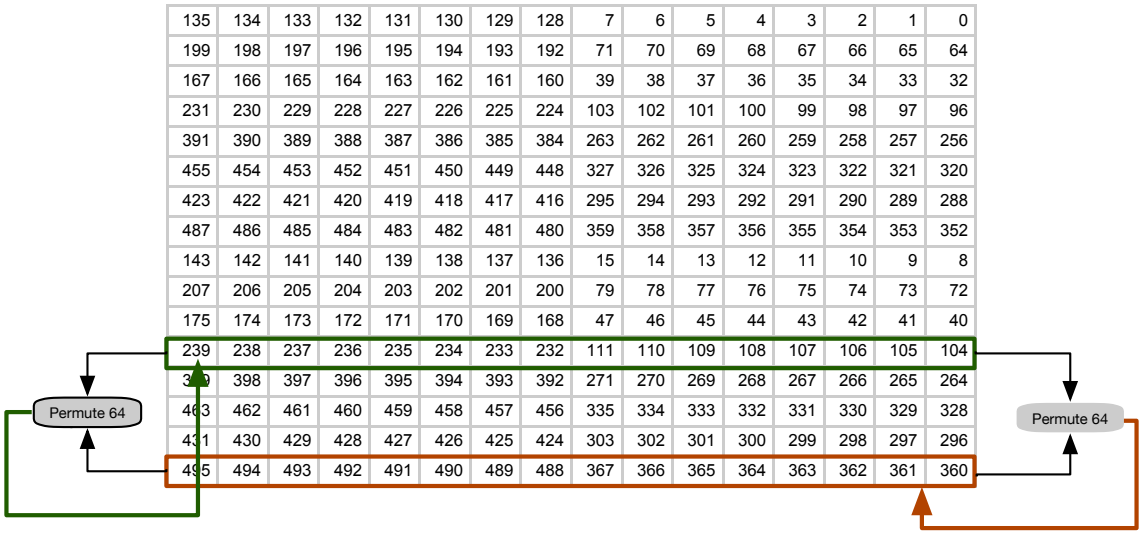


Figure 4.10: Our new transpose algorithm where $i = 3$.

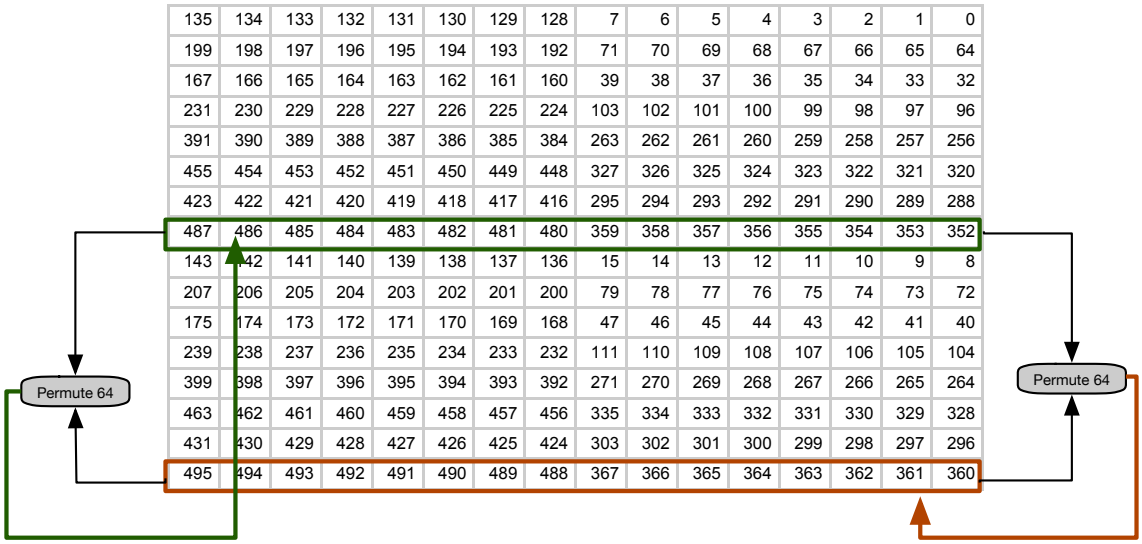


Figure 4.11: Our new transpose algorithm where $i = 7$.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
79	78	77	76	75	74	73	72	71	70	69	68	67	66	65	64
47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
111	110	109	108	107	106	105	104	103	102	101	100	99	98	97	96
271	270	269	268	267	266	265	264	263	262	261	260	259	258	257	256
335	334	333	332	331	330	329	328	327	326	325	324	323	322	321	320
303	302	301	300	299	298	297	296	295	294	293	292	291	290	289	288
367	366	365	364	363	362	361	360	359	358	357	356	355	354	353	352
143	142	141	140	139	138	137	136	135	134	133	132	131	130	129	128
207	206	205	204	203	202	201	200	199	198	197	196	195	194	193	192
175	174	173	172	171	170	169	168	167	166	165	164	163	162	161	160
239	238	237	236	235	234	233	232	231	230	229	228	227	226	225	224
399	398	397	396	395	394	393	392	391	390	389	388	387	386	385	384
463	462	461	460	459	458	457	456	455	454	453	452	451	450	449	448
431	430	429	428	427	426	425	424	423	422	421	420	419	418	417	416
495	494	493	492	491	490	489	488	487	486	485	484	483	482	481	480

Figure 4.12: Our new transpose algorithm, final result.

4.4.1 Evaluation

To evaluate the performance of our proposed algorithm for matrix transpose,we implemented and benchmarked the following variations of the transpose function on M2. Note that all these variations are implemented as a function that transposes 1024 32-bit integers.

- Our algorithm: the implementation of our algorithm.
- scalar_Clang: the implementation of the algorithm in Listing 9 compiled by the Clang compiler. Our investigation shows that code inside the function is unrolled three times.
- scalar_GCC: the implementation of the algorithm in Listing 9 compiled by the GCC compiler.
- AVX-512: the implementation of the algorithm in Listing 10.

The result of this experiment is shown in Figure 4.13. As can be seen, our algorithm is four times faster than all other implementation. Also note that Clang optimization to unroll the loop three times leads to better performance.

Moreover, we combined our transpose algorithm with our new prefix sum function described in Section 4.2.1. The performance of this combination is the same as the performance of the transpose function without prefix sum. This result indicates that if a transpose function is necessary to be used, the best possible way to use it is to explicitly

combine it with the transpose function with one of the operators of a whitebox compression, as the transpose function is the bottleneck. Note that to combine the transpose function with an operator, the operator implementation must be completely unrolled.

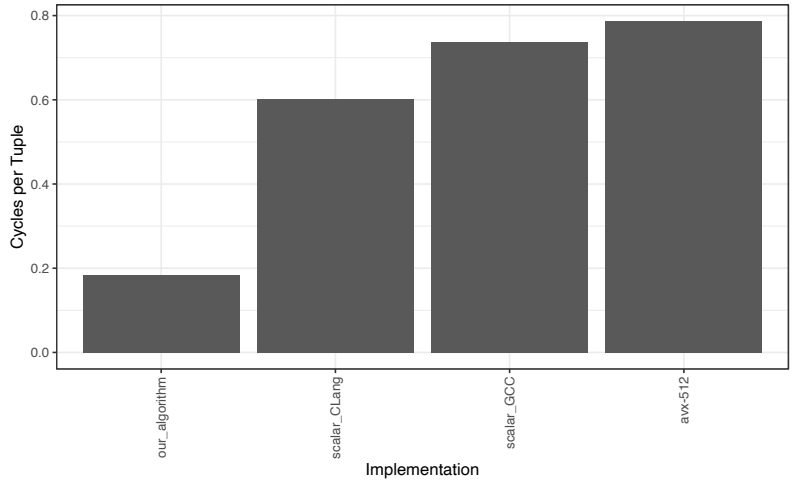


Figure 4.13: The result of different implementations of the transpose algorithm.

4.5 Load

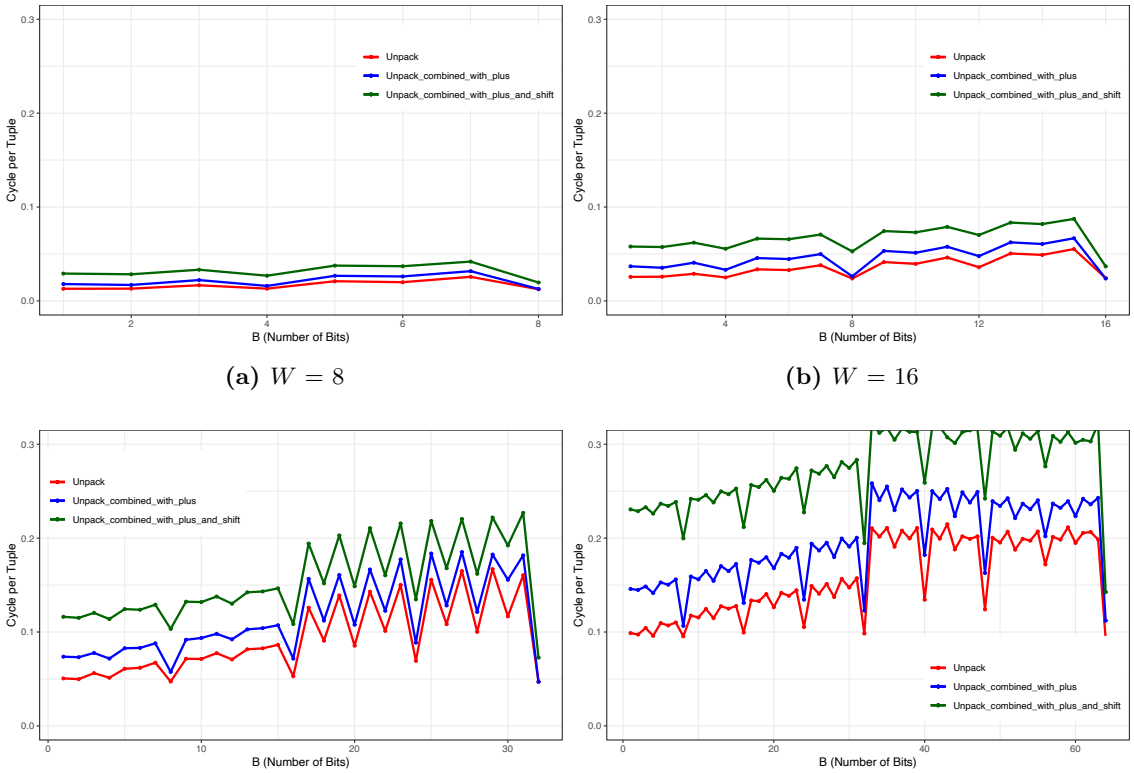
The load function takes a vector of 1024 offsets and dictionary base as input and fetches the actual values pointed by these offsets into the result vector. Also, the load function can take a vector of 1024 absolute pointers as input and fetches the actual values pointed by these pointers into the result vector. The load function leads to executing additional instructions to load data from memory and store them back, which could be avoided if the load function is combined with the bit-unpacking function. Moreover, the load function could be implemented using GATHER instruction. As shown in the transpose function, the GATHER and SCATTER instructions do not lead to better performance than scalar code. However, the scale parameter in the GATHER instruction could be used to generate 64-bit pointers from offsets without additional computation.

4.6 Shift

The shift function takes a vector of 1024 tuples and a integer n as input and shifts every tuple of vector, n bits to the left. The shift function leads to executing additional instruc-

tions to load data from memory and to store them back, which could be avoided if the load function is combined with the bit-unpacking function.

4.6.1 Evaluation



(a) $W = 8$ **(b)** $W = 16$

(c) $W = 32$ **(d)** $W = 64$

Figure 4.14: The performance of the bit-unpack function combined with shift and plus vs the bit-unpack function combined with the plus vs the bit-unpack function in terms of cycles per tuple.

In this experiment, we benchmarked the bit-unpack function's performance in three cases: when it is combined with plus and shift, when it is combined with plus, and when it is separate. Figure 4.14 shows the results of this experiment performed on M2. As can be seen, when the bit-unpacking is combined with shift and plus, the result is 2 times worse. Therefore, unlike plus, we propose to not mix the shift function with bit-packing and plus. If the shift function is needed, a separate library of a combination of bit-unpacking, plus, and the shift is required.

4.7 Patch

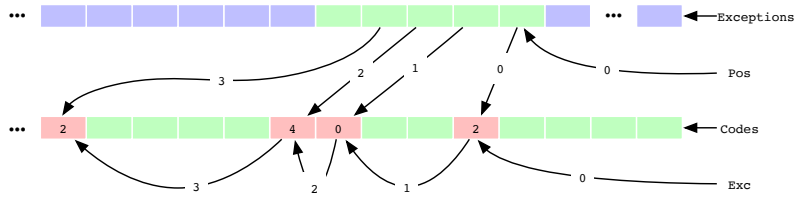


Figure 4.15: An example of the patch function.

The Patch function takes a vector of bit-unpacked integers (*code vector*), a vector of exceptions (*exception vector*), a 16-bit integer (*exc*), and 16-bit (*pos*) as input and outputs a 1024-tuple vector, where exceptions are put into their correct position inside the code vector. Figure 4.15 illustrates an example of this function. As can be seen, the *exc* parameter points to the first exception inside the code vector, while the *pos* parameter points to the starting point of the exception vector corresponding to this code vector. From the starting point, the exceptions are placed back into the *code vector* at the position denoted by $cursor = exc$. Note that before each replacement, the *cursor* value is updated with the *code[cursor]* which points to the next exception location in the code vector. Moreover, since bit-unpacking is combined with plus, the base value must be subtracted from the cursor value.

```
1    void patch(ANY *__restrict__ codes, ANY *__restrict__ exceptions,
2    uint16_t exc, uint16_t pos, ANY diff) {
3        int next, cur = exc;
4        exceptions += pos;
5        for (int i = 0; cur < 1024; i++, cur = next) {
6            next = cur + codes[cur] + 1 - diff;
7            codes[cur] = exceptions[i];
8        }
9    }
```

Listing 11: The implementation of the patch function.

The implementation of the patch function is shown in Listing 11. As can be seen, in Line 6, the offset of the next exception is calculated, while in Line 7, the value of the current exception is placed in the code vector.

Note that in the whitebox model, the patch function needs to neutralize the bit-unpacking function's effect if the bit-unpacking function is combined with other functions. For ex-

ample, if the bit-packing is combined with plus, the base parameter is added to all bit-unpacked data, actual tuples, and offset. Therefore, the base value needs to be subtracted from the offset to restore the original offset in the patch function.

4.7.1 Evaluation

To evaluate the performance of the patch function, we implemented and benchmarked the patch function on M2. Note that all these variations are implemented as a function that split 32-bit integers.

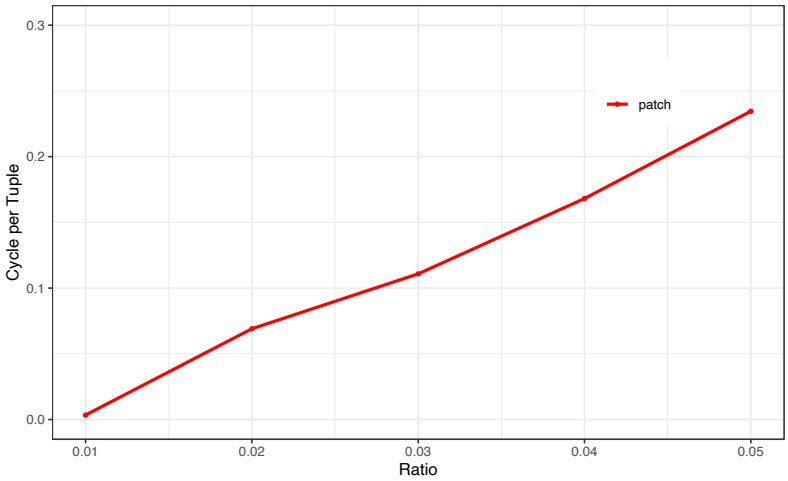
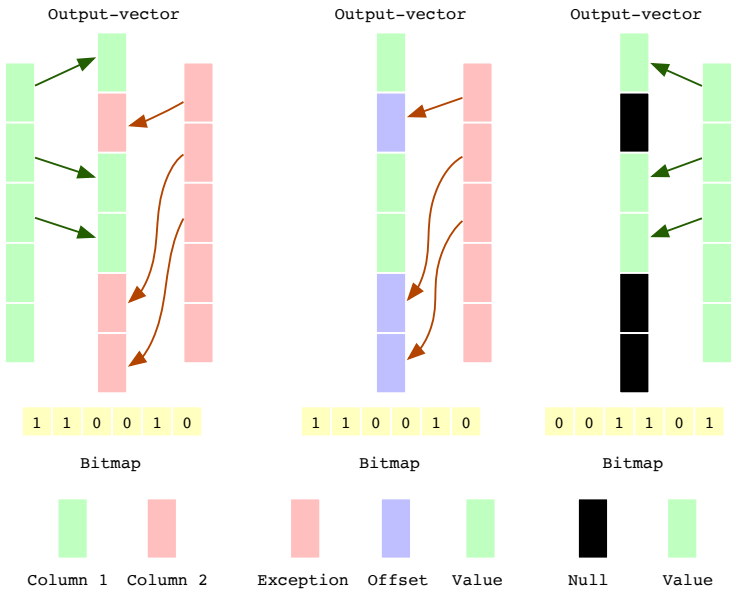


Figure 4.16: The result of the patch function.

The result of this experiment is shown in Figure 4.16. The X-axis shows the ratio of the number of exceptions in 1024 tuples. Note that the process of choosing exceptions is done entirely randomly and the experiment is repeated 10 times. The Y-axis shows the median of cycles used per tuple for 10 experiments. As can be seen, our implementation has a reasonable performance for all cases.

4.8 Split

The split function takes two vectors of arbitrary size (in total 1024 tuples) and a 1024-bits bitmap as input and outputs a 1024-tuple vector, which combines two given columns based on given the bitmap. Figure 4.17 A illustrates an example of this function. As can be seen, for each tuple in the output vector, the corresponding bit in the bitmap array specifyes from which vector the tuple should be copied.



(a) Split **(b)** Patch **(c)** NULL

Figure 4.17: An example of all variations of the split function.

Besides combining columns, the split function has several other use cases which are described in the following sections.

4.8.1 Patch substitution

The patch function is not the right candidate for the PFOR scheme with bit-widths below 5, as it introduces too many compulsory exceptions and thus degrading the compression ratio. To overcome this problem, the patch function could be replaced with the split function that places exceptions into their position specified by the bitmap. This change forces the PFOR scheme to have a bitmap of size 1024 for bit-widths below 5, while there is no need to have any compulsory expectations. However, as the exceptions are not frequent in the patching encoding, the split function could be optimized to handle a few exceptions. Figure 4.17 B illustrates an example of this function.

4.8.2 Null support

Null tuples do not contain any specific information and could be removed from a column to achieve better compression. Later, the original column could be reconstructed using the split function, which inflates the vector of tuples by putting actual values in their original position using the provided bitmap. Note that for this case of the split function,

only one column is needed as the second column contains only Null values. Figure 4.17 C illustrates an example of this function.

4.8.3 Implementation and Evaluation

The split function is implemented using `if-else` conditions, as shown in Listing 12. For each tuple, the corresponding bit is extracted. If the bit is set to 1, the tuple from the first column is copied to the result vector, otherwise the tuple from the second column. This implementation could be optimized further if there exists a pattern that shows how data is distributed over two columns. However, in most cases, the distribution is random.

```
1   void split(ANY *__restrict col1, ANY *__restrict col2,
2           ANY *__restrict result,
3           uint64_t *__restrict bitmap) {
4       int counter1 = 0;
5       int counter2 = 0;
6       uint64_t tmp_bitmap;
7       uint64_t tmp;
8       for (size_t i = 0; i < 16; ++i) {
9           tmp = bitmap[i];
10          for (int j = 0; j < 64; ++j) {
11              tmp_bitmap = (tmp >> j) & (1);
12              if (tmp_bitmap == 1) {
13                  result[i * 64 + j] = col1[counter1++];
14              } else {
15                  result[i * 64 + j] = col2[counter2++];
16              }
17          }
18      }
19  }
```

Listing 12: The scalar implementation of the split function.

This implementation is expected to have a bad performance as it introduces many branches that cannot be predicted by a CPU. To mitigate this problem, the EXPAND_LOAD instructions can be used instead. Listing 13 shows an example of this implementation for tuples of type unsigned 32-bit int. In Lines 9-10, a SIMD register is filled with tuples from both columns. Note that the tuples from the second column are placed in their position inside the register using the bit-flipped bitmask.

The result of performance of the different implementations of the split functions is shown in Figure 4.18. The X-axis shows the ratio of the number of tuples from the first column to 1024 tuples. Note that the process of dividing tuples among two columns is done entirely

```
1   void split512(uint32_t *__restrict col1,
2       uint32_t *__restrict col2, uint32_t *__restrict result,
3                 uint64_t *__restrict bitmap) {
4       __m512i tmp;
5       uint16_t *bitmap_ = reinterpret_cast<uint16_t *>(bitmap);
6       uint16_t tmp_bitmap;
7       int tmp_count;
8       for (int k = 0; k < 64; ++k) {
9           tmp_bitmap = bitmap_[k];
10          tmp = _mm512_maskz_expandloadu_epi32(tmp_bitmap, col1);
11          tmp = _mm512_mask_expandloadu_epi32(tmp, ~tmp_bitmap, col2);
12          tmp_count = __builtin_popcount(tmp_bitmap);
13          col1 += tmp_count;
14          col2 += 16 - tmp_count;
15          _mm512_storeu_si512(result + k * 16, tmp);
16      }
17  }
```

Listing 13: The SIMD implementation of the split function.

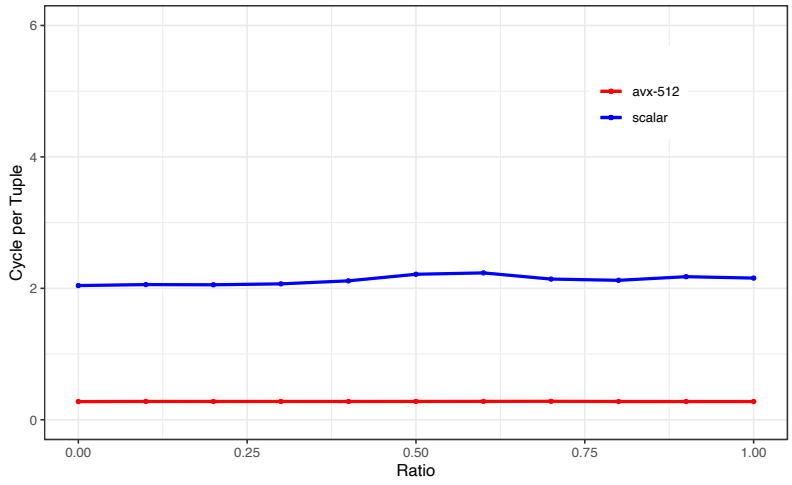


Figure 4.18: The result of different implementations of the split function.

randomly. As can be seen, our SIMD implementation improves the performance 10 times. Also it can be concluded that the ratio does not have any effect on both implementations.

To implement the split function, which simulates the patch function, it is crucial to consider that only a few tuples inside the result vector are required to be replaced. To leverage this property, a pruning technique can be used in which a block of tuples that are not needed to be replaced are skipped. Listing 14 shows the implementation of this

function. In Line 6, the pruning technique is applied for a block of size 64. Note that the block size could have a value of 64, 32, 16, or 8. If no tuple inside a block is an exception, the bitmask corresponding to this block is 0 and could be skipped. If a bitmask is not 0, the most significant bit with value 1 is extracted (Line 7). The position of this bit could be easily found using the CTZLL instruction. This position could be used to place an exception in its right place in the code vector. After the replacement, this bit is set to 0, making it possible to find the next bit that is set to 1 in the next iteration.

```
1   void split_patch_with_pruning(ANY *codes, ANY *exception, uint64_t *bitmap) {
2       int counter = 0;
3       uint64_t bitset;
4       for (size_t k = 0; k < 16; ++k) {
5           bitset = bitmap[k];
6           while (bitset != 0) {
7               uint64_t t = bitset & -bitset;
8               int r = __builtin_ctzll(bitset);
9               codes[k * 64 + r] = exception[counter++];
10              bitset ^= t;
11          }
12      }
13  }
```

Listing 14: The scalar implementation of the split-patch function.

If the bitmap corresponding to a block is not 0, instead of finding the positions of bits set to 1, the EXPLAND_LOAD instruction can be used to put the exceptions in their positions, as shown in Listing 15. Moreover, the split-patch function could be implemented without the pruning technique. This implementation is shown in Listing 16.

To evaluate the performance of the split-patch function, we implemented and benchmarked the six mentioned variations of the split-patch function on M2. Note that all these variations are implemented as a function that splits 32-bit integers.

The result of this experiment is shown in Figure 4.19. The X-axis shows the ratio of the number of tuples from the first column to 1024 tuples. Note that the process of dividing tuples among two columns is done entirely randomly. As can be seen, our scalar implementation with the block size of 64 results in the best possible performance.

To implement the split function, which restores the original layout of null-compressed data, the pruning technique and the EXPAND_LOAD instructions can be used. To evaluate the performance of different implementations of the split-null function, we implemented and benchmarked all possible variations of the split function on M2 listed as follows. Note that all these variations are implemented as a function that split 32-bit integers.

```
void split_patch_with_pruning_avx512(uint32_t *codes,
        uint32_t *exception, uint16_t *bitmap) {
    __m512i codes_;
    int count = 0;
    uint16_t bitmap_ = 0;
    for (int i = 0; i < 64; i++) {
        bitmap_ = *(bitmap + i);
        if (bitmap_ != 0) {
            codes_ = _mm512_load_si512(codes + i * 16);
            codes_ = _mm512_mask_expandloadu_epi32(codes_, bitmap_, exception + count);
            _mm512_store_si512(codes + i * 16, codes_);
            count += __builtin_popcount(bitmap_);
        }
    }
}
```

Listing 15: The SIMD implementation of the split-patch function using the pruning technique.

```
void split_patch_avx512(uint32_t *codes, uint32_t
        *exception, uint16_t *bitmap) {
    __m512i codes_;
    int count = 0;
    uint16_t bitmap_ = 0;
    for (int i = 0; i < 64; i++) {
        bitmap_ = *(bitmap + i);
        codes_ = _mm512_load_si512(codes + i * 16);
        codes_ = _mm512_mask_expandloadu_epi32(codes_, bitmap_, exception + count);
        _mm512_store_si512(codes + i * 16, codes_);
        count += __builtin_popcount(bitmap_);
    }
}
```

Listing 16: The SIMD implementation of the split-patch function.

- The implementation of the split-null function without the pruning technique, shown in Listing 17.
- The implementation of the split-null function with the pruning technique, shown in Listing 18.
- The SIMD implementation without the pruning technique using AVX-512 instructions, shown in Listing 19.
- The SIMD implementation with the pruning technique using AVX-512 instructions, shown in Listing 20.

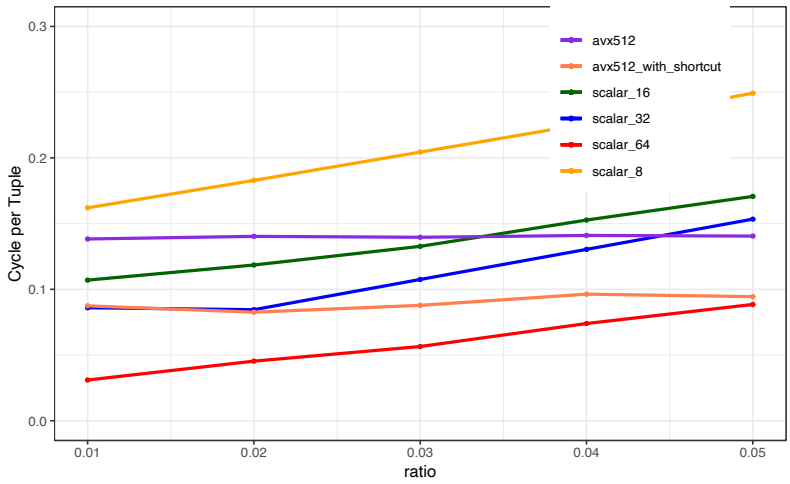


Figure 4.19: The result of different implementations of the split function for the patch case.

- The SIMD implementation without the pruning technique using SSE instructions, shown in Listing 21. In this implementation, the EXPAND_LOAD instructions are simulated by using 128-bit SHUFFLE instructions provided in SSE. The SHUFFLE instruction needs another vector that defines where each byte of the register should go. For this purpose, a table is already generated that can be accessed by an 8-bit integer as index. For each possible 8-bit index, the table contains 128-bit data needed for the SHUFFLE instruction. Note that this implementation also can be used for implementing the regular split function.

```
1  void null(uint32_t *data, uint32_t *result, uint64_t *bitmap) {
2      int counter = 0;
3      uint64_t bitset;
4      for (size_t k = 0; k < 16; ++k) {
5          bitset = bitmap[k];
6          for (int i = 0; i < 64; i++) {
7              uint64_t t = bitset & -bitset;
8              int r = __builtin_ctzll(bitset);
9              result[k * 64 + r] = data[counter++];
10             bitset ^= t;
11         }
12     }
13 }
```

Listing 17: The scalar implementation of the split-null function.

```
void null_with_shortcut(uint32_t *data, uint32_t *result, uint64_t *bitmap) {
    int counter = 0;
    uint64_t bitset;
    for (size_t k = 0; k < 16; ++k) {
        bitset = bitmap[k];
        while (bitset != 0) {
            uint64_t t = bitset & -bitset;
            int r = __builtin_ctzll(bitset);
            result[k * 64 + r] = data[counter++];
            bitset ^= t;
        }
    }
}
```

Listing 18: The scalar implementation of the split-null function using the pruning technique.

```
void split_null_avx512(uint32_t *data, uint32_t
*result, uint16_t *bitmap) {
    __m512i result_;
    int count = 0;
    uint16_t bitmap_ = 0;
    for (int i = 0; i < 64; i++) {
        bitmap_ = *(bitmap + i);
        result_ = _mm512_loadu_si512(result + i * 16);
        result_ = _mm512_mask_expandloadu_epi32(result_, bitmap_, data + count);
        _mm512_storeu_si512(result + i * 16, result_);
        count += __builtin_popcount(bitmap_);
    }
}
```

Listing 19: The SIMD implementation of the split-null function.

```
void split_null_with_shortcut_avx512(uint32_t *data, uint32_t *result, uint16_t *bitmap) {
    __m512i result_;
    int count = 0;
    uint16_t bitmap_ = 0;
    for (int i = 0; i < 64; i++) {
        bitmap_ = *(bitmap + i);
        if (bitmap_ != 0) {
            result_ = _mm512_loadu_si512(result + i * 16);
            result_ = _mm512_mask_expandloadu_epi32(result_, bitmap_, data + count);
            _mm512_storeu_si512(result + i * 16, result_);
            count += __builtin_popcount(bitmap_);
        }
    }
}
```

Listing 20: The SIMD implementation of the split-null function using the pruning technique.

```
void null_sse(uint32_t *data, uint32_t *result, uint64_t *bitmap) {
    __m128i *result_ = reinterpret_cast<__m128i *>(result);
    int counter = 0;
    __m128i vecA;
    __m128i vecB;
    __m128i vecC;
    __m128i vecD;
    __m128i tmp;
    for (int i = 0; i < 16; ++i) {
        uint64_t w = bitmap[i];
        for (int k = 0; k < 4; ++k) {
            uint8_t byteA = (uint8_t) w;
            uint8_t byteB = (uint8_t) (w >> 4);
            w >>= 8;
            uint8_t byteC = (uint8_t) w;
            uint8_t byteD = (uint8_t) (w >> 4);
            w >>= 8;
            vecA = _mm_load_si128(reinterpret_cast<const __m128i *>(lu_table[byteA]));
            vecB = _mm_load_si128(reinterpret_cast<__m128i *>(lu_table[byteB]));
            vecC = _mm_load_si128(reinterpret_cast<const __m128i *>(lu_table[byteC]));
            vecD = _mm_load_si128(reinterpret_cast<__m128i *>(lu_table[byteD]));
            tmp = _mm_loadu_si128((const __m128i *) (data));
            tmp = _mm_shuffle_epi8(tmp, vecA);
            _mm_storeu_si128(result_++, tmp);
            data += __builtin_popcount(byteA);
            tmp = _mm_loadu_si128(reinterpret_cast<__m128i *>(data));
            tmp = _mm_shuffle_epi8(tmp, vecB);
            _mm_storeu_si128(result_++, tmp);
            data += __builtin_popcount(byteB);
            tmp = _mm_loadu_si128((const __m128i *) (data));
            tmp = _mm_shuffle_epi8(tmp, vecC);
            _mm_storeu_si128(result_++, tmp);
            data += __builtin_popcount(byteA);
            tmp = _mm_loadu_si128((const __m128i *) (data));
            tmp = _mm_shuffle_epi8(tmp, vecD);
            _mm_storeu_si128(result_++, tmp);
            data += __builtin_popcount(byteA);
        }
    }
}
```

Listing 21: The SIMD implementation of the split-null function for SSE.

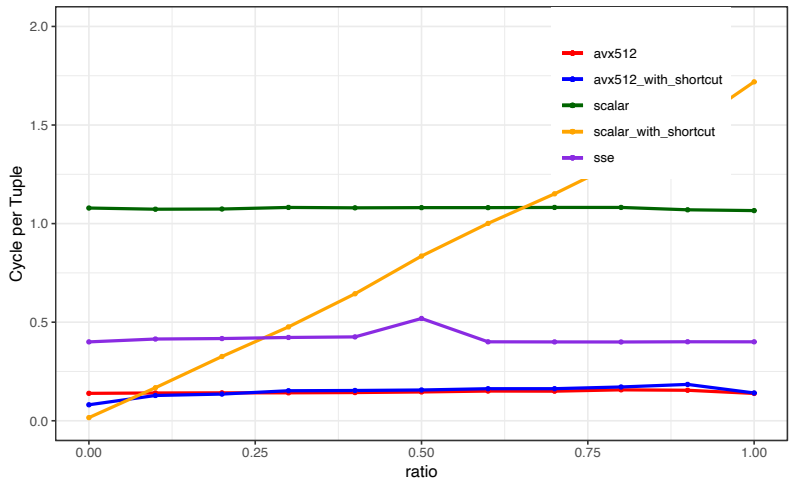


Figure 4.20: The result of different implementations of the split function for the Null case.

The result of this experiment is shown in Figure 4.19. The X-axis shows the ratio of the number of tuples from the first column to 1024 tuples. Note that the process of dividing tuples among two columns is done entirely randomly. As can be seen, our SIMD implementation with AVX-512 instructions results in the best possible performance. Moreover, the SSE performance is better than the scalar version, which shows that the SSE implementation can be used for CPUs supporting AVX2 or SSE instructions.

4.9 RLE

The RLE function takes an RLE-compressed column and restores the original representation of tuples. As described in Section 2.7, the layout of RLE is usually interleaved in a way that for each run, the length and the actual value are adjacent. This leads to several problems. Firstly, to further compress the already RLE-compressed data, a bit-packing technique can be used. However, the interleaved layout makes it nearly impossible to efficiently bit-pack length values as they are paired with the actual values, and the best number of bits used to bit-pack length values are affected by the number of bits used to bit-pack the actual values. For example, the value type could be of type string or a large integer that does not let values with a small length to efficiently bit-packed. Secondly, the interleaved layout makes it hard to expose the RLE layout to the higher-level operations since the separation of length values from actual values needs to be done in that operator. Finally, the implementation of the traditional RLE is block-based as the number of length

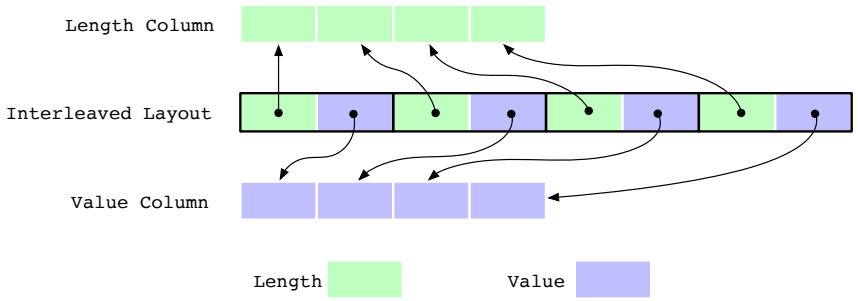


Figure 4.21: An example of our RLE layout where the length and value columns are stored separately.

and value pairs for each vector are not specified. This enforces the decompression of the whole page instead of a vector during random access.

To solve these problems, we propose a new layout for RLE, in which the length and the actual values are separated from each other and stored in two different columns: the Length column and the Value column. An example of this layout could is shown in Figure 4.21. Furthermore, to determine which length and value pair belongs to which vector, an integer value, cnt_i, for every vector is added to this layout, which shows the number of length and value pairs until the beginning of vector i. Note that all cnts are stored in a mini-frame. Moreover, the actual number of length and value pairs for each vector could be calculated as $cnt_{i+1} - cnt_i$.

4.9.1 SIMD RLE Layout

Furthermore, to SIMDize the decompression of our new RLE layout, we propose a new SIMD RLE layout that leverages our transposed layout discussed in Section 4.2.1. To explain our new RLE layout, consider an example of 64 values instead of 1024 values, shown in Figure 4.25. Note that each color indicates a unique value. In our new layout, tuples inside each column are separately compressed to value and length pairs. After that, all these pairs are sorted based on the row number of the first element of a run. If this value is equal for two runs, they are sorted based on the first element's column number. This layout results in copying the first element of all columns into the value vector, as shown in Figure 4.25. For simplicity, we name the first list, which contains the first tuples of every column, fixed-list, and the list that comes after this list, variable-list.

However, in a non-tranposed layout, for a vector with high similarity, data is compressed into few length and value pairs. In contrast, in our SIMD RLE layout, in addition to those

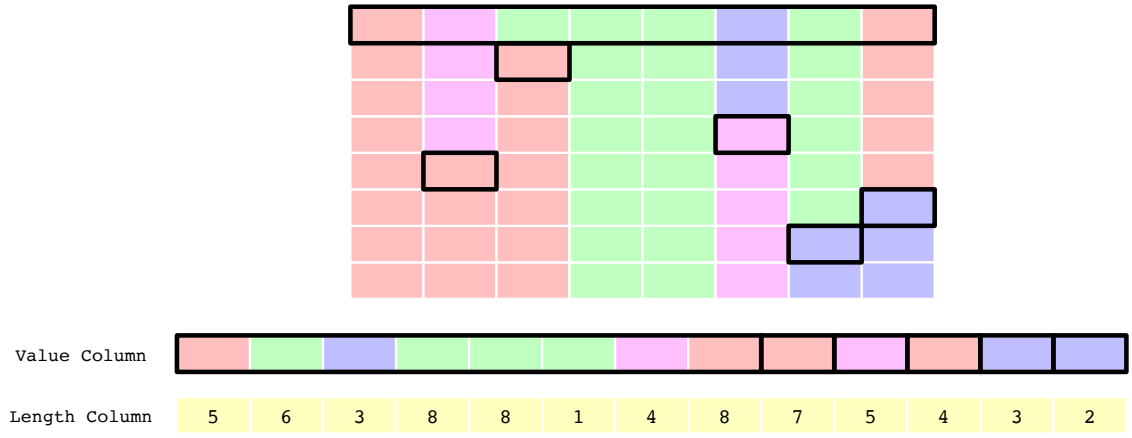


Figure 4.22: An example of SIMDizing our new RLE layout.

pairs, the first elements of all columns also need to be copied into the value vector together with their corresponding length into the length vector. To mitigate this problem, for these cases, a simple interleaved RLE layout can be used to compress the N first elements of both value and length columns, where $N = 1024/W$.

4.9.2 Intermediate Representation

The proposed SIMD RLE layout leads to a worse compression ratio compared to the non-transposed layout when the number of runs is small. As mentioned in the previous section, one solution is to compress the first elements of both value and length vectors using the simple RLE. However, this solution only solves the overhead of the SIMD RLE layout when the number of runs is very small. This is due to the fact that the value vector needs to be compressed inside the value vector, which leads to using a value-size length parameter.

To solve the overhead problem of the SIMD layout, we propose an intermediate RLE layout for cases where the number of runs is smaller than 32 or 64, dependent on number of bits used to represent the bit-packed value. This layout is illustrated in Figure 4.23. The green rectangles represent tuples with the same value. As can be seen, the green box is repeated 21 times. In the non-transposed RLE layout, this run is represented by a length and value pair. In our layout, we represent this run as a pair of value and length, where the length is divided into 4 parameters:

- Col: this parameter determines the number of columns that starts with the corresponding value.
- L: this parameter determines the number of tuples in the last column with the corresponding value.
- R: this parameter determines the number of tuples in the right-side of the group of columns starts with the corresponding value.
- Pos: this parameter determines the position of value and length pair which needs to copied after the first N elements, where value is equal to the value and length is equal to R.

Note that the pos parameters of all pairs which are not needed to be copied variable-size list are assigned to the value of the pos parameter of last value in variable-size list.

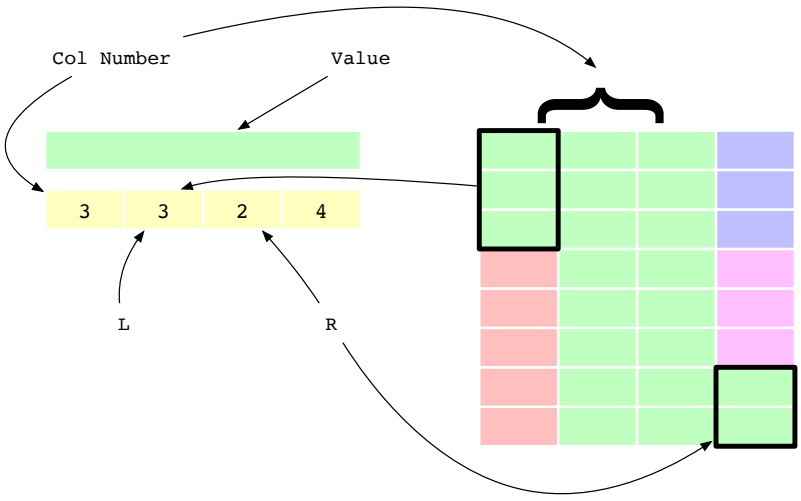


Figure 4.23: The layout of intermediate RLE.

An example of this layout is given in Figure 4.24. As can be seen, 4 runs exist. For the first run, the value is 5, while the length is 504. In the intermediate layout, the length of this run is converted to 16, 24, 0, 3 which means:

- 16 columns are started with this value.
- this value is repeated 24 times in the last column that belongs to this run.
- 0 tuples are in the right side of columns that are started with this value.
- it needs to be placed in position $N + 3$.

Note that the first run does not need to be copied into the length value pairs that come after the N pairs. However, to minimize the computation and repetition overhead, it needs to be copied. To redo this copy, the pos value is assinged to 3. Later, this pair is replaced by the last value length pair, which also has a pos of value 3.

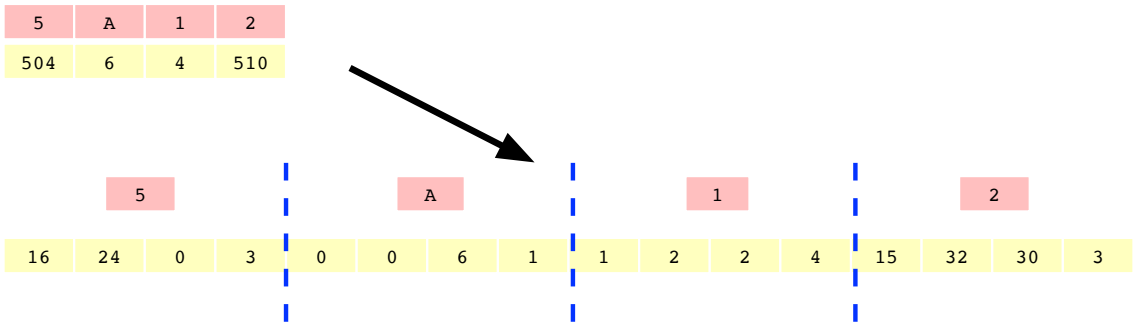


Figure 4.24: An example of the intermediate RLE layout.

To convert this layout to the RLE SIMD layout, we propose the following implementation that is shown in Listing 22. In this implementation, for each pair, first, we copy the value and length of the pair into fix-list, *Col* times, starting from the offset. Note that the last pair length needs to be set to parameter *L*. Then, we copy this value to the variable-list at the position defined by the parameter *Pos*. The length of this pair is defined by the *R* parameter. When the number of runs is higher than 8, this implementation leads to branch miss predictions. For these cases, we propose the following implementation, which is the unrolled version of the previous one. In this implementation, as shown in Listing 23, we repeat coping the length and value pairs to the fixed-list and variable list 32 times.

```
void
convert_intermediate_representation_to_simd_layout(
uint32_t *value, uint32_t *length, int counter, uint32_t *r_value,
                                            uint32_t *r_length) {
    int length_tmp;
    int offset = 0;
    for (int i = 0; i < counter; ++i) {
        length_tmp = length[i];
        for (int j = 0; j < length_tmp; ++j) {
            r_value[offset + j] = value[i];
            r_length[offset + j] = length[4 * i];
        }
        offset += length_tmp;
        r_value[32 + length[4 * i + 3]] = value[i];
        r_length[32 + length[4 * i + 3]] = length[4 * i + 2];
    }
}
```

Listing 22: The implementation of the convert function when number of runs are below 8.

```
void
convert_intermediate_representation_to_simd_layout_unrolled(
        uint32_t *value, uint32_t *length, int counter, uint32_t *r_value,
        uint32_t *r_length) {
    uint32_t length_tmp = length[0];
    int offset = 0;
    int count = 0;
    for (int i = 0; i < 32; ++i) {
        r_length[i] = length[offset * 4 + 2];
        r_value[i] = value[offset];
        offset += (length_tmp & 0U);
        count = (length_tmp == 0) ? 0 : count + 1;
        length_tmp = length[offset] - count;
        r_length[i] = 31;
    }
    for (int i = 0; i < counter; ++i) {
        r_length[32 + length[i * 4 + 3]] = length[i * 4 + 2];
        r_value[32 + length[i * 4 + 3]] = value[i];
    }

}
```

Listing 23: The implementation of the convert function when number of runs are between 8 and 32.

4.9.3 Decompression Algorithm

Our SIMD decompression algorithm is dependant on the MASK_EXPANDLOAD instructions introduced for AVX-512 registers. Listing 25 shows an example of this algorithm implemented for `int` values. As can be seen in Lines 4-8, first, all N elements of the value and length columns are loaded into four registers. Moreover, two registers containing the constant values 0 and 1 are generated in Lines 12-13. During each iteration, Lines 17-39, first, the registers containing the actual values are stored at the output location. Then, the length vectors containing the length values are compared against the SIMD register containing the value 0, and then the value 1 is subtracted from length values. Note that all length values l_i, are stored as $l_i - 1$. The previous operation results is a bitmask, which indicates lengths with value 0, by setting a bit to 1. If the bitmasks are not zero, all length values are higher than 0, and in the next iteration, the value vectors need to be stored. However, if the bitmask is not equal to 0, the corresponding value is replaced using the new value with the MASK_EXPANDLOAD operation. Furthermore, the offset pointing to the beginning of length and value columns are incremented by the number of bits set to 1 in the bitmask. This number is calculated using the POPCOUNT instruction.

```
1   void simple_rle(int *__restrict value,
2   int * __restrict length,
3   int * __restrict output, uint8_t counter) {
4       int length_tmp, offset = 0;
5       for (int i = 0; i < counter; ++i) {
6           length_tmp = length[i] + 1;
7           for (int j = 0; j < length_tmp; ++j) {
8               output[offset + j] = value[i];
9           }
10          offset += length_tmp;
11      }
12  }
```

Listing 24: The implementation of the RLE function for `32-bit int`.

```
void simd_rle(int *__restrict value, int *__restrict length,
int * __restrict output, uint8_t counter) {
    // load from the value column
    __m512i _value1 = _mm512_loadu_si512(value);
    __m512i _value2 = _mm512_loadu_si512(value + 16);
    // load from the length column
    __m512i _length1 = _mm512_loadu_si512(length);
    __m512i _length2 = _mm512_loadu_si512(length + 16);
    // output
    __m512i *_output = reinterpret_cast<__m512i *>(output);
    // two 512-bit simd registers with all lanes set to 0 and 1
    __m512i _zero = _mm512_set1_epi32(0);
    __m512i _one = _mm512_set1_epi32(1);
    // 64 bit mask
    uint64_t mask1 = 0, mask2 = 0;
    int offset = 32;
    for (int i = 0; i < 31; i++) {
        // store the value register
        _mm512_storeu_si512(_output + (2 * i) + 0, _value1);
        _mm512_storeu_si512(_output + (2 * i) + 1, _value2);
        // if length = 1, set bit
        mask1 = _mm512_cmp_epi32_mask(_length1, _zero, 0);
        mask2 = _mm512_cmp_epi32_mask(_length2, _zero, 0);
        // length = length - 1
        _length1 = _mm512_sub_epi32(_length1, _one);
        _length2 = _mm512_sub_epi32(_length2, _one);
        if (mask1 != 0 || mask2 != 0) {
            // update
            _value1 = _mm512_mask_expandloadu_epi32(_value1, mask1, value + offset);
            _length1 = _mm512_mask_expandloadu_epi32(_length1, mask1, length + offset);
            offset += __builtin_popcount(mask1);
            _value2 = _mm512_mask_expandloadu_epi32(_value2, mask2, value + offset);
            _length2 = _mm512_mask_expandloadu_epi32(_length2, mask2, length + offset);
            offset += __builtin_popcount(mask2);
        }
    }
    _mm512_storeu_si512(_output + (31 * 2) + 0, _value1);// store the last one
    _mm512_storeu_si512(_output + (31 * 2) + 1, _value2);// store the last one
}
```

Listing 25: The SIMD implementation of the RLE function for `32-bit int`.

```
1   void naive_simd_rle(int *__restrict value,
2   int *__restrict length,
3   int * __restrict output, uint8_t counter) {
4       int length_tmp, offset = 0;
5       __m512i tmp;
6       for (int i = 0; i < counter; ++i) {
7           length_tmp = length[i] + 1;
8           for (int j = 0; j < length_tmp; j += 16) {
9               tmp = _mm512_set1_epi32(value[i]);
10              _mm512_storeu_si512(output + j + offset, tmp);
11          }
12          offset += length_tmp;
13      }
14  }
```

Listing 26: The naive SIMD implementation of RLE function for `32-bit int`.

4.9.4 Evaluation

To evaluate the performance and compression ratio of our proposed layout, we implemented the following variations of RLE. Note that all these variations are implemented as a function that returns 1024 32-bit integers.

- simd: the implementation of the SIMD RLE layout shown in Listing 25.
- simd2: the implementation of the SIMD RLE layout, which uses another RLE compression for the first N items.
- scalar: the scalar implementation of the RLE layout shown in Listing 24.
- simple_simd : the implementation of the RLE layout shown in Listing 26.
- intermediate : the implementation of the RLE layout. which uses intermediate layout.

To benchmark the RLE performance in terms of the number of cycles spent per tuple and compression ratio, two factors need to be considered: the number of runs and each run position. The first factor shows how many times two adjacent tuples in a vector are different or how many runs exist, while the second one shows where this change occurs in the vector.

To benchmark the compression ratio, in our experiment, we generated a vector with random tuples such that D adjacent values are different, while the position of this difference is random. For D with values between 1 and 250, we calculated the number of bits used by each RLE variation. Note that the assumption is that the tuples can be bit-packed to 16 bit tuples. Moreover, the scalar RLE has the same layout as the simple_simd; thus, it is removed from the benchmark.

Figure 4.25 shows the result of the experiment. The Y-axis indicates the number of bits used for compression, while the X-axis shows the number of runs. As can be seen, our RLE SIMD layout compression ratio is worse than simple RLE where $D < 120$. However, for cases with $D < 50$, this could be compensated with the intermediate RLE layout. For $D > 120$, we can see that our layout provides better compression. It is also worth mentioning that each run has a unique value in our experiment, which is the worst case for SIMD RLE. This is due to the fact that if they are not unique, the first N elements possible could be compressed better as there is more similarity between them.

Figure 4.26 shows the result of the previous experiment, where the Y-axis is replaced by the compression ratio. The more detailed version of this result is shown in Figure 4.27.

The previous experiment was repeated 10 times to benchmark the decompression speed, and for each variation, the medians of 10 cycles per tuple are calculated. Figure 4.28 shows

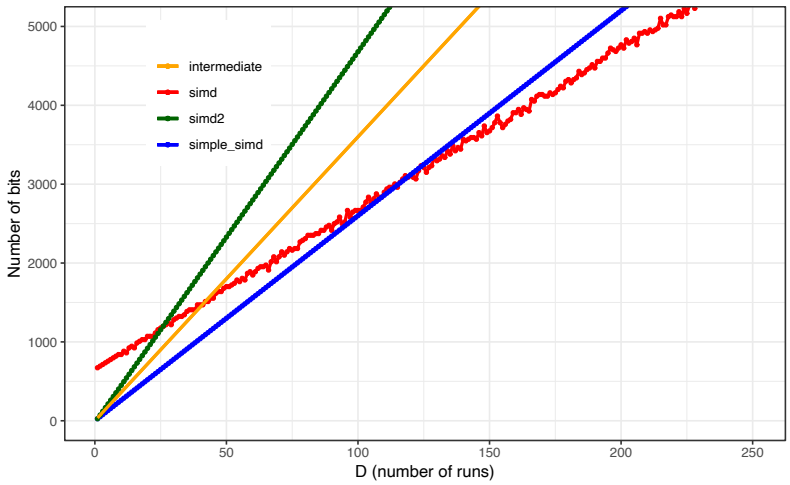


Figure 4.25: The result of different layouts for RLE in terms of number of bits used to compress data for different number of runs.

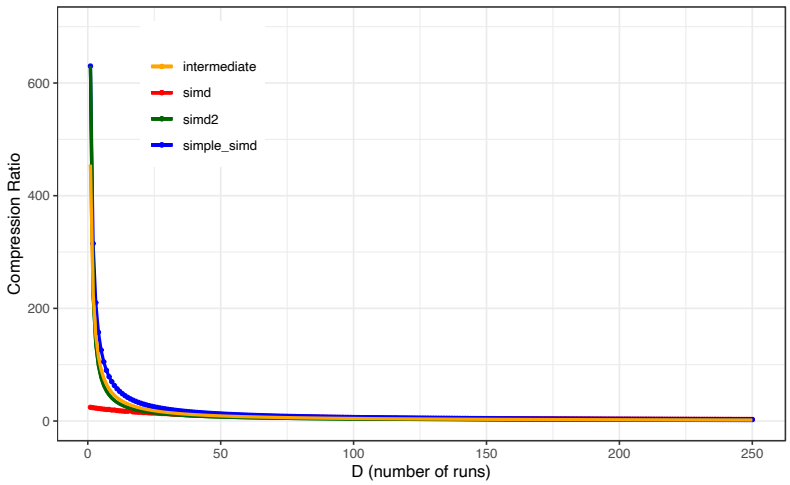


Figure 4.26: The result of different layouts for RLE in terms of compression ratio for different number of runs.

the result. The result shows that for $D < 9$, all layouts provide the same decompression speed. For other values of D, the result is entirely different for each case. The scalar implementation is not comparable to others, and the simple_simd is only comparable to SIMD RLE, where $D < 50$. For $D > 50$, our SIMD layout provides nearly 3 times faster decompression speed. Moreover, the decompression speed is constant for $D > 60$, as the value and length registers need to be updated during each iteration. Note that the computation cost of converting the intermediate layout to RLE is not considered as for

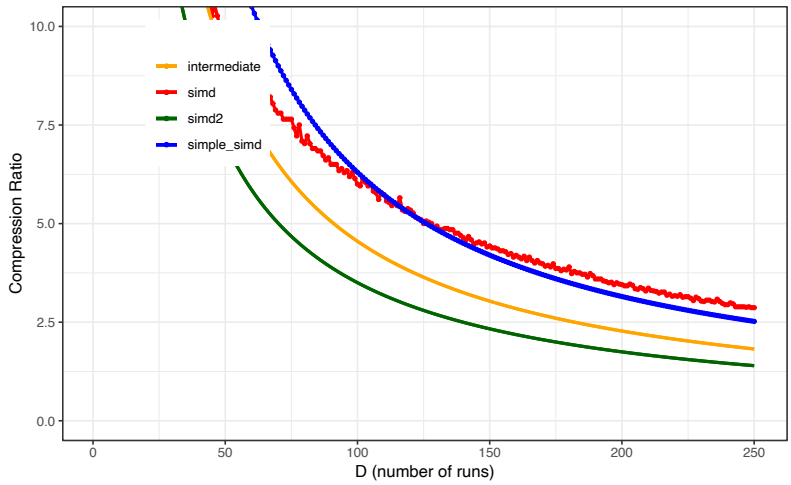


Figure 4.27: The result of different layouts for RLE in terms of compression ratio for different number of runs.

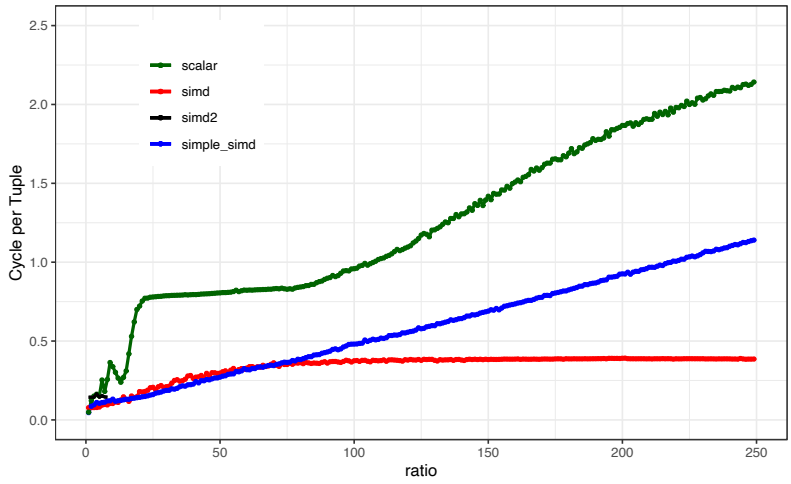


Figure 4.28: The performance of different layouts for RLE in terms of number of cycle per tuple for different number of runs.

cases below 8, it is between 0.01 and 0.06 cycles per tuple, and for cases above 8 it is 0.06 cycles per tuple.

Another possible option to improve the compression ratio for small values of D is to use a none-transposed RLE layout. However, during decompression, the transpose function should be added to simple_simd, which adds the latency of 0.18 cycle per tuple. This idea results in the same compression ratio as a regular RLE layout while having nearly two times slower decompression speed (nearly 0.3 cycles per tuple).

5

Composable Compression Schemes

The result from the previous chapter shows that complete whiteboxing of compression function leads to performance overhead. For example, the plus function results in the best possible performance when it is combined with the bit-unpacking function. In this chapter we propose the composable compression model, a variation of the whitebox compression model, that allows more complex functions favoring the decompression speed. In the composable compression model, we follow two principles:

1. The patching phase needs to be separated from other parts of decomporession into a single function.
2. If it is possible, the decomposed functions, except for patching, should be combined with the bit-unpacking function. This results in avoiding extra load and store instructions (materialization) for each separated function.

In this chapter, we describe the composable version of the compression schemes such as PFOR, PDelta, PFOR-Delta and PDICT. To simulate a blackbox compression scheme in the composable compression model, decompression functions should be transformed into a composition of one or many functions according to the composable model principles. This change makes the decompression flexible and recursive. Note that all required functions in this section are already explained in Section 4.

5.1 Composable PFOR

The composable version of PFOR (C-PFOR) can be defined as the composition of the bit-unpack combined with the plus and patch functions, as shown in Figure 5.1. In Figure 5.1 the green boxes show data and metadata needed for C-PFOR, while the red boxes show

the functions or operators use to simulate the blackbox PFOR. In the first step, data is bit-unpacked, and the base value is added to the data. The next step is patching, which puts exceptions in their right position inside the result vector. Note that the *diff* parameter is subtracted from all offsets as they are increased by the previous function. Moreover, based on the bit-width, the *exc*, *pos*, *diff* metadata could be replaced by a bitmap. In that case, instead of the patch function, the split-patch function is called.

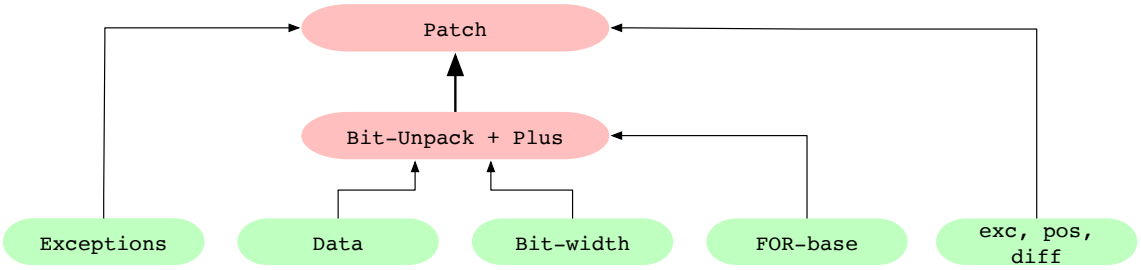


Figure 5.1: The function composition for C-PFOR.

In contrast to all the PFOR variants, the C-PFOR uses a different base for every 1024 tuples. This change makes our scheme more flexible and leads to a better compression ratio. This is due to the fact that the value of bit-width is dependent on the base value, and the best possible base value could be different for each chunk, while in the previous versions of PFOR, the base value is fixed for all chunks.

5.1.1 Evaluation

To evaluate the effect of having a different base and bit-width for each chunk in C-PFOR, we designed an experiment where a sample of 2^{20} integers is extracted from all the 32-bits non-nullable integer columns of all datasets in the Public BI benchmark (3). Then, these samples are compressed with C-PFOR in two scenarios, A and B. In scenario A, for each sample, a single best value b and *base* is determined, while in scenario B, for every 1024 tuples, the best value for b and *base* is determined. Note that the best value for b and *base* is determined using the algorithm proposed by Heman (42). Then, the compression ratio is calculated. The difference between the compression ratios is shown in Figure 5.2. Each point shows the difference between the two scenarios for a given dataset. The detailed version of this figure is represented in Table 2. As can be seen in Figure 4.1, the difference is significant, and indeed, in some cases, the compression ratio is two times

higher, which shows having a different base and bid-width is effective. Moreover, the average of compression ratio in scenario A is 2.895563, while in scenario B it is 3.498244, which is another proof of the effectiveness of C-PFOR.

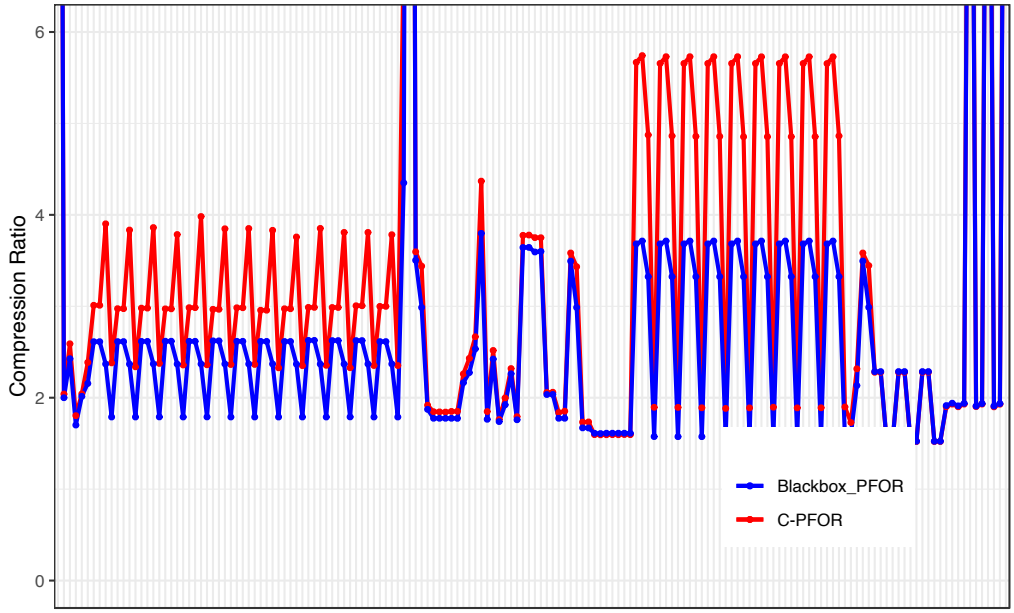


Figure 5.2: Difference between scenario A and B in terms of compression ratio.

5.2 Composable PDelta

In this section, we discuss the composable variant of PDelta (C-PDelta), which can be defined as the composition of the bit-unpack, patch and prefix sum functions, as shown in Figure 5.3. In the first step, data is bit-unpacked. Note that the bit-unpack function can be replaced by bit-unpack combined with the plus, where the value 0 is added to tuples. This change results in less code as there is no need to have separate functions for bit-unpacking and bit-unpacking combined with plus. Moreover, only having the combination of both functions does not lead to a significant perforamnce penalty, as shown in Section 4.3. The next step is the patching function, which is used in a different order compared to C-PFOR. Finally, the prefix sum function, in which the prefix sum of all data is calculated with the order defined in the Section 4.2.1. This step can be replaced with the combination of prefix sum and transposed functions to keep the order of tuples

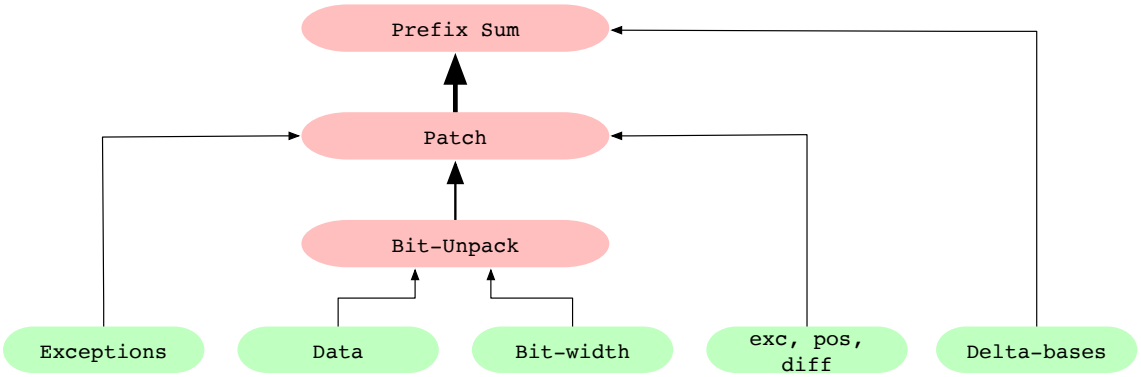


Figure 5.3: The function composition for composable PDelta.

5.3 Composable PFOR-Delta

In this section we discuss the composable variant of PFOR-Dleta (C-PFOR-Delta), which can be defined as the composition of the bit-unpack combined with plus, patch adn prefix sum functions, as shown in Figure 5.4. The C-PFOR-Delta scheme works similarly to C-PDelta, except that a base value is added to all bit-unpacked tuples in the first function.

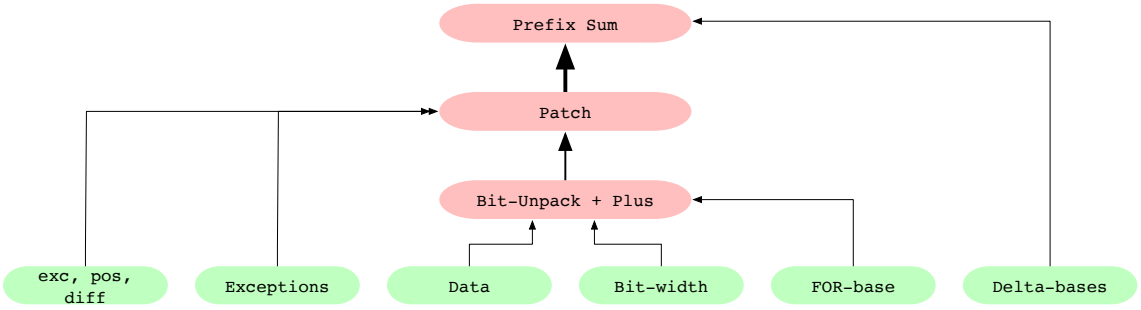


Figure 5.4: The function composition for composable PFOR-Delta.

5.4 Composable PDICT

In this section, we discuss the composable variant of PDICT (C-PDICT), which can be defined as the composition of the bit-unpack, load, and patch functions. These functions can be ordered in two different ways; we refer to them as variation 1 and 2. In variation

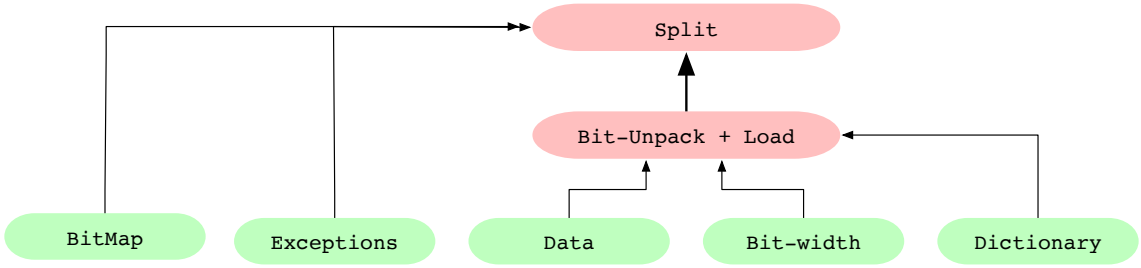


Figure 5.5: The function composition of C-PDICT, variation 1.

1, as shown in Figure 5.5, the first step is the combination of the bit-unpack function and load functions. Note that each compressed tuple represents an offsets to the dictionary list. After the first step, the data is fetched from the dictionary, which are the actual tuples in case of numeric data types or an offset in case of strings. The next step is the patch function, which is replaced by the split function. This is because, after loading, each tuple presents data from the dictionary and the patching offsets are not valid anymore.

In variation 2, as shown in Figure 5.6, compared to variation 1, the patch function is in the middle of decompression. First, a vector of absolute pointers (64-bit unsigned integer) is calculated as a result of the bit-unpacking combined with shift. Second, the patch function visits the exception positions and replaces the address with the exception address. Third, the values are fetched into the result vector using the pointers in the load function.

The variation 1 results in a faster scheme because, first, there is no need to have a separate load function, leading to extra load and store instruction for each tuple, and secondly, there is no need to do the shift operation. Moreover, in terms of compression ratio, the variation 1 only has the overhead of 1 bit per tuple as a bitmap needs to be used for the split function.

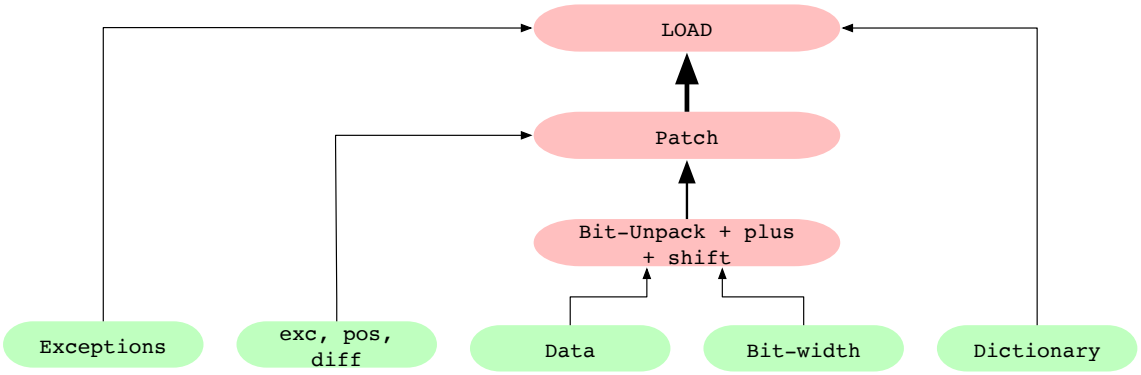
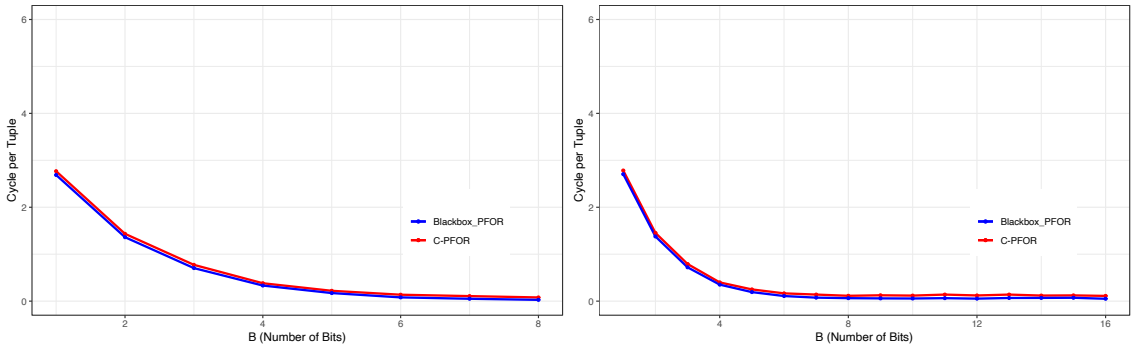


Figure 5.6: The function composition of C-PDICT, variation 2.

5.4.1 Evaluation

Supporting black-box compression schemes in the composable compression model might introduce overhead as instead of executing one function, the expression tree of smaller functions needs to be evaluated and executed.

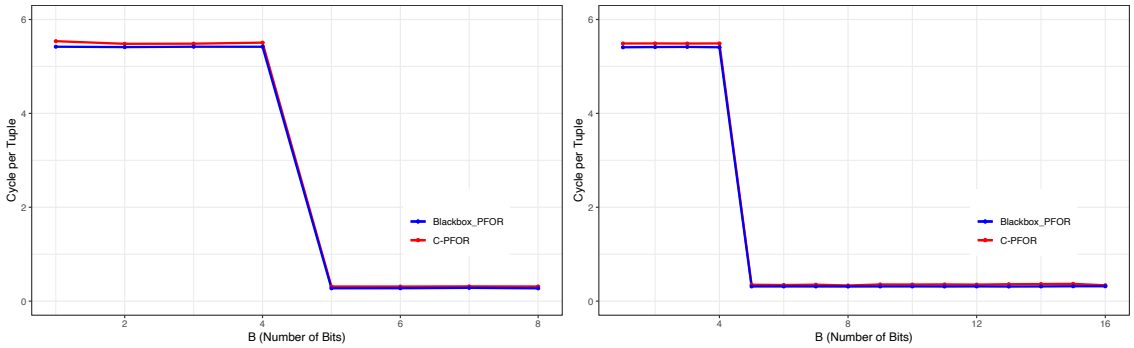
To evaluate the effect of this overhead, we implemented a simple expression tree. Leaves of this tree represent required data and metadata, while non-leaf nodes express a function. As an exmaple, the C-PFOR scheme is given to this tree. Besides that, the functions used in C-PFOR are combined into one function to resemble blackbox compression. We benchmarked the performance of both cases for the 0.05% and 0% exceptions ratios. The result is shown in Figure 5.7 and 5.8. As can be seen, the difference is not significant, and indeed, both are very similar, proving that converting a blackbox scheme to the composable model does not lead to performance degradation.



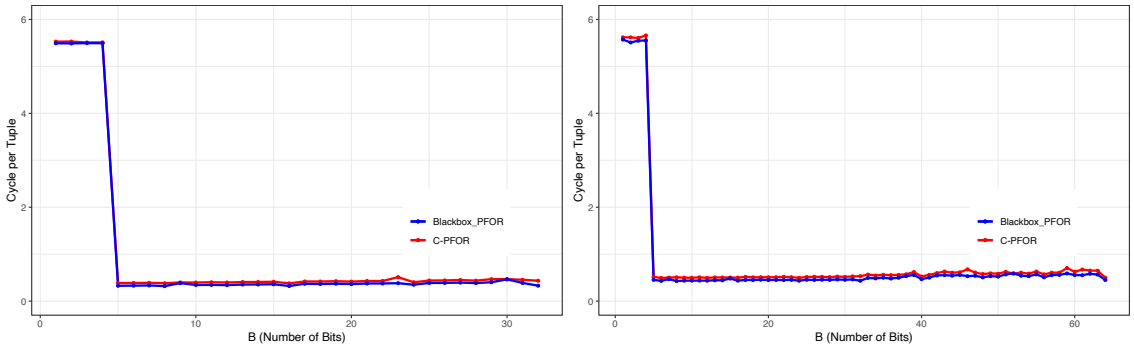
(a) $W = 8$ **(b)** $W = 16$

(c) $W = 32$ **(d)** $W = 64$

Figure 5.7: The performance of C-PFOR vs PFOR in terms of cycles per tuple for each possible B and W. The exception ratio is 0.



(a) $W = 8$ **(b)** $W = 16$



(c) $W = 32$ **(d)** $W = 64$

Figure 5.8: The performance of C-PFOR vs PFOR in terms of cycles per tuple for each possible B and W. The exception ratio is 0.05.

6

Conclusions

In this thesis, we built the foundation of a new file format for big data, by focusing on two key components, namely the SIMD-friendly and composable compression. We first reviewed the state-of-the-art big data file formats and discussed their shortcomings. Then, we briefly discussed how a new data file format could overcome the shortcomings of state-of-the-art big data file formats and be compatible with new trends emerging in the software ecosystem, such as shifting from HDFS to the Cloud environment, new metadata, and CPU capabilities.

Our main contribution in this thesis is that we made compression schemes such as PFOR, PDelta, PDICT, and RLE SIMD-friendly and composable. Being SIMD-friendly allows these compression schemes to exploit the widest SIMD register supported by a CPU (even future CPUs), and being composable allows them to be flexible and be recursively combined. Moreover, we showed that the overhead of having these compression schemes as a composable compression is negligible.

We also introduced an alternative mechanism to handle outliers in light-weight compression schemes, namely the split function. In addition to the design and implementation of a completely SIMD-friendly bit-unpacking, we made the bit-packing more flexible by having different bit-widths for each vector. We showed that the overhead of this change is negligible and, indeed, in some cases, the performance is even improved. This change also leads to better compression ratio for all schemes which are dependant on bit-unpacking. Finally, we significantly improved the compression ratio of C-PFOR, in some cases, even 2 times, by storing the base parameter for every vector. This change also makes our new scheme more flexible.

6.1 Research questions

In this section we revisit the research questions that were defined in Section 1.5.

Research Question 1. How can compression functions be decomposed to multiple functions to be flexible and efficient at the same time? is it possible to apply the whitebox compression principles to blackbox compression schemes to achieve the desired flexibility without performance penalty?

Our result shows that the complete whiteboxing of compression schemes such as PFOR, PDelta, PDICT, despite providing the desired flexibility, leads to the performance penalty as tuples need to be materialized (load/store) during each function, except for the functions used for outlier handling. To solve this, we proposed a composable compression model, a variation of whitebox compression, based on two new principles. First, the patch function needs to be separated. Second, if possible, the other parts of decompression need to be combined with the bit-unpacking function to avoid extra materialization. In this new model, we defined C-PFOR as the composition of the bit-unpack combined with the plus and patch functions, C-PDelta as the composition of the bit-unpack, prefix sum, and patch functions, and the C-PDICT as the composition of the bit-unpack combined with the load, and patch functions.

Research Question 2. The patching technique, which was proposed to mitigate the light-weight compression vulnerability to outliers, leads to a worse compression ratio in cases where only a few bits are required to represent data. What is a good alternative to the patching technique to handle outliers in these cases?

We have introduced the split technique as an alternative to the patching techniques. In our split technique, a bitmap, which contains one bit for each tuple, denotes whether a tuple needs to be replaced with the corresponding exception. Furthermore, we have optimized the split technique using the pruning technique. During the replacement, 64 tuples are skipped at once if there is no exception among them, which is the case for most tuples as exceptions happen rarely. Our benchmark shows that our new implementation can achieve a comparable compression ratio compared to the patch function for a low number of exceptions, and even better performance for a higher number of exceptions.

Research Question 3. How can we make compression schemes such as PFOR, PDelta, PDICT and RLE SIMD-friendly in which the compression scheme can exploits the instructions for the widest register that a CPU supports, both for the current available CPUs and CPUs with wider SIMD registers in the future?

To have composable compression, which is SIMD-friendly, we defined each function as follows. Note that 1024 is used as a prototype.

Bit-unpacking: This function bit-unpacks all numeric data. To exploit SIMD instructions of any kind, we designed a new layout called the 1024-bits interleaved layout in which for every 1024 bits of data, the order of tuples are changed. Our experiment on different CPUs with different SIMD register sizes shows that the performance of our layout grows linearly with the register size. Note that our layout is capable of supporting the new SIMD registers that will be released in the future. Furthermore, the scalar implementation of our layout is improved 2 times using the multi cursor technique. Our results show that using explicit SIMD instructions is worth the effort as the result of compiler auto-vectorization, in some cases, is 2 times slower. Finally, we investigated the idea of having an intermediate word size to bit-unpack, which was not successful. We may consider investigating this idea further in the future, as this idea depends on CPU instructions, which could be accelerated in the future.

Plus: This function adds a constant to all the tuples of a vector of numeric values. Our findings show that it is necessary to combine this function with the bit-unpacking function to achieve the best possible performance. This is because there is no need to load and store tuples again in the plus function.

Prefix Sum: This function calculates the prefix sum of the tuples in a vector. To be able to SIMDize this function, we needed to remove the data dependency of the prefix sum operation. Data dependency means to be able to process a tuple, the result of the previous tuples needs to be known, which leads to a massive performance penalty. To remove the data dependency, we designed a new layout called the transposed layout that changes the order of tuples so that there is no data dependency between SIMD lanes. This layout results in 20 times performance improvement compared to a simple (non-SIMD implementation) and 10 times compared to the state-of-the-art SIMD implementation. Note that the overhead of this layout is one bit per tuple.

Transpose: the transpose function restores the original layout of tuples, or transforms a vector to the transposed layout. To be able to SIMDize this function, the new SCATTER instruction introduced for AVX-512 registers was not useful. Therefore, we proposed a new algorithm to do the transpose operator, which uses 17 registers and is not dependant on the SCATTER instruction. This algorithm results in 4 times better performance compared to other possible implementations of the transpose function.

Split This function combines two vectors. The performance of the split function is improved 10 times with our new implementation, which uses the new SIMD instruction

EXPAND_LOAD. Moreover, we used the split function instead of the patch function in C-PFOR to solve the compulsory exception problem of PFOR. The compulsory exception problem happens in PFOR when the number of bits used to bit-pack data is small, and many tuples need to be considered as exception resulting in a worse compression ratio. Finally, we used the split function to decompress nullable columns and improved its performance using SIMD instructions.

RLE This function decompresses an RLE-compressed vector, which is compressed using our layout, where the value and length are decoupled and stored in two different vectors. To be able to SIMDize this function, we proposed a new layout called SIMD RLE in which we RLE-compress each SIMD lane separately. This results in having a better compression ratio and 3 times faster decompression speed for cases where the number of runs is high. However, for other cases, the overhead of the compression ratio and the decompression speed is significant. To mitigate this problem, we designed an intermediate RLE layout, which later is converted to our SIMD RLE layout.

Research Question 4. Does our new SIMD-friendly compression schemes need to be implemented using explicit SIMD intrinsics? Is auto vectorization offered by compilers, such as GCC, able to obtain the performance of our explicit implementation?

To answer this question, all defined composable functions needs to be considered. For bit-unpacking, our experiments show that the auto-vectorized code by GCC or Clang compilers does not match the implementation in which explicit intrinsics are used. The GCC auto vectorized code in all experiments is 3-4 times slower than explicit implementation. The Clang compiler can achieve the same performance as the explicit implementation only in some cases. For example, on a CPU with AVX-512, the result of the code auto-vectorized by the Clang compiler is in some cases two times worse. On a CPU with AVX2, for 64-bit integers, when the number of bits is more than 32, the code auto-vectorized by Clang compiler completely fails to vectorize the code. It is worth mentioning that even in cases where both approaches result in the same performance, a specific version of a compiler is used. Therefore, it is not guaranteed that for other versions, the compilers can achieve the same performance. Overall, depending on auto-vectorization can lead to inconsistency in the performance in different systems. Therefore we suggest using explicit code as it guarantees to achieve the best performance. In terms of maintenance we only need to maintain the code generator and not the actual code. Note that the bit-unpacking code is generated by a simple code generator that generates the code using SIMD intrinsics during compilation for the widest SIMD register the target CPU supports.

For other functions such as the transpose function, the algorithms are dependant on specific instructions and cannot be implemented using the scalar instruction. However, as the implementations are only a few lines, it does not increase the complexity of the system.

6.2 Future Work

In this thesis, we showed that how the composable compression model could be efficiently implemented using SIMD instructions. The SIMD-friendly composable compression is the basis for a new big data file format. In the future, we plan to work on other aspects of a file format such as supporting metadata, tuple deletion, and allowing nested table structures in metadata and data.

References

[1] Anastasia Ailamaki, David Dewitt, Mark Hill, and Marios Skounakis. **Weaving Relations for Cache Performance**. 09 2001. v, 14, 15, 35

[2] Benjamin Schlegel, Rainer Gemulla, and Wolfgang Lehner. **Fast integer compression using SIMD instructions**. pages 34–40, 01 2010. v, 16, 17

[3] Bogdan Guita, Diego Tomé, and Peter Boncz. **White-box Compression: Learning and Exploiting Compact Table Representations**. 01 2020. v, 4, 11, 30, 31, 55, 98

[4] Yin Huai, Ashutosh Chauhan, Alan Gates, Gunther Hagleitner, Eric Hanson, Owen O'Malley, Jitendra Pandey, Yuan Yuan, Rubao Lee, and Xiaodong (Frank) Zhang. **Major technical advancements in Apache Hive**. 06 2014. v, 33, 34

[5] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter Boncz, Thomas Neumann, and Alfons Kemper. **Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation**. 06 2016. vi, 10, 35, 36

[6] **Apache Spark**. `https://spark.apache.org/`. 1

[7] **Apache Hive**. `https://hive.apache.org/`. 1

[8] **Apache Impala**. `https://impala.apache.org/`. 1

[9] **Apache Parquet**. `http://parquet.apache.org/`. 1, 2

[10] **Apache ORC**. `https://orc.apache.org/`. 1

[11] Sacheendra Talluri, Alicja undefineduszczak, Cristina L. Abad, and Alexandru Iosup. **Characterization of a Big Data Storage Workload in**

the Cloud. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, ICPE '19, page 33–44, New York, NY, USA, 2019. Association for Computing Machinery. 1

[12] Stefan Heule, Marc Nunkesser, and Alexander Hall. **HyperLogLog in Practice: Algorithmic Engineering of a State of the Art Cardinality Estimation Algorithm**. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, page 683–692, New York, NY, USA, 2013. Association for Computing Machinery. 1

[13] Harald Lang, Andreas Kipf, Linnea Passing, Peter Boncz, Thomas Neumann, and Alfons Kemper. **Make the most out of your SIMD investments: counter control flow divergence in compiled query pipelines**. pages 1–8, 06 2018. 1

[14] Oueslati Wided and Jalel Akaichi. **A Survey on Data Warehouse Evolution**. *International Journal of Database Management Systems*, **2**, 11 2010. 1

[15] Ibrahim Hashem, Ibrar Yaqoob, Nor Anuar, Salimah Mokhtar, Abdullah Gani, and Samee Khan. **The rise of "Big Data" on cloud computing: Review and open research issues**. *Information Systems*, **47**:98–115, 07 2014. 1

[16] Marcin Zukowski, Niels Nes, and Peter Boncz. **DSM vs. NSM: CPU performance tradeoffs in block-oriented query processing**. pages 47–54, 01 2008. 2, 13

[17] George P. Copeland and Setrag N. Khoshafian. **A Decomposition Storage Model**. In *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data*, SIGMOD '85, page 268–279, New York, NY, USA, 1985. Association for Computing Machinery. 2

[18] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. **Column-Stores vs. Row-Stores: How Different Are They Really?** In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, page 967–980, New York, NY, USA, 2008. Association for Computing Machinery. 2

[19] Anastassia Ailamaki, David J. DeWitt, and Mark D. Hill. **Data Page Layouts for Relational Databases on Deep Memory Hierarchies**. *The VLDB Journal*, **11**(3):198–215, November 2002. 2

[20] Alen Stojanov, Ivaylo Toskov, Tiark Rompf, and Markus Püschel. **SIMD Intrinsics on Managed Language Runtimes**. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, page 2–15, New York, NY, USA, 2018. Association for Computing Machinery. 2

[21] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. **Super-Scalar RAM-CPU Cache Compression**. In *Proceedings of the 22nd International Conference on Data Engineering*, ICDE '06, page 59, USA, 2006. IEEE Computer Society. 3

[22] Peter Boncz, Viktor Leis, and Thomas Neumann. **Fast Static Symbol Table (FSST): fast text compression that allows random access**. 2020. 3

[23] **apache/parquet-format**. `https://github.com/apache/parquet-format/blob/master/src/main/thrift/parquet.thrift`. 3

[24] Biswapesh Chattopadhyay, Priyam Dutta, Weiran Liu, Ott Tinn, Andrew McCormick, Aniket Mokashi, Paul Harvey, Hector Gonzalez, David Lomax, Sagar Mittal, Roee Aharon Ebenstein, Nikita Mikhaylin, Hung ching Lee, Xiaoyan Zhao, Guanzhong Xu, Luis Antonio Perez, Farhad Shahmohammadi, Tran Bui, Neil McKay, Vera Lychagina, and Brett Elliott. **Procella: Unifying serving and analytical data at YouTube**. *PVLDB*, **12(12)**:2022–2034, 2019. 3

[25] Bogdan Guita, Diego Tomé, and Peter Boncz. **White-box Compression: Learning and Exploiting Compact Table Representations**. 01 2020. 4

[26] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. **The Hadoop Distributed File System**. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, page 1–10, USA, 2010. IEEE Computer Society. 4

[27] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanaël Prémillieu, Alastair Reid, Alejandro Rico, and Paul Walker. **The ARM Scalable Vector Extension**. *CoRR*, **abs/1803.06185**, 2018. 9, 39

[28] George P. Copeland and Setrag N. Khoshafian. **A Decomposition Storage Model**. In *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data*, SIGMOD '85, page 268–279, New York, NY, USA, 1985. Association for Computing Machinery. 14

[29] Daniel Abadi, Samuel Madden, and Miguel Ferreira. **Integrating compression and execution in column-oriented database systems**. pages 671–682, 01 2006. 14, 19, 28

[30] **OLAP Council White Paper**. 2004. 14

[31] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter A. Boncz. **Super-Scalar RAM-CPU Cache Compression**. In Ling Liu, Andreas Reuter, Kyu-Young Whang, and Jianjun Zhang, editors, *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, page 59. IEEE Computer Society, 2006. 16, 20, 27, 28

[32] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Prof. Dr. Alexander Zeier, and Jan Schaffner. **SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units.** *PVLDB*, **2**:385–394, 08 2009. 16, 17

[33] Orestis Polychroniou and Kenneth A. Ross. **Efficient Lightweight Compression Alongside Fast Scans**. In *Proceedings of the 11th International Workshop on Data Management on New Hardware*, DaMoN'15, New York, NY, USA, 2015. Association for Computing Machinery. 17, 19

[34] Thomas Willhalm, Ismail Oukid, Ingo Müller, and Franz Färber. **Vectorizing Database Column Scans with Complex Predicates**. 08 2013. 17

[35] Daniel Lemire and Leonid Boytsov. **Decoding billions of integers per second through vectorization**. *Software: Practice and Experience*, **45**, 01 2015. 17, 23, 24, 39, 57, 59

[36] Yinan Li and Jignesh Patel. **BitWeaving: Fast scans for main memory data processing**. pages 289–300, 06 2013. 18, 19

[37] David Huffman. **A method for the construction of minimum-redundancy codes**. *Resonance*, **11**:91–99, 02 2006. 19

[38] Ian Witten, Ian H, Neal, Radford M, Cleary, and John G. **Arithmetic Coding for Data Compression**. *Communications of the ACM*, **30**:520–, 01 1987. 19

[39] En-Hui Yang and J.C. Kieffer. **Efficient universal lossless data compression algorithms based on a greedy sequential grammar transform. I. Without context models**. *Information Theory, IEEE Transactions on*, **46**:755 – 777, 06 2000. 19

[40] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. **Compressing Relations and Indexes**. In Susan Darling Urban and Elisa Bertino, editors, *Proceedings of the Fourteenth International Conference on Data Engineering, Orlando, Florida, USA, February 23-27, 1998*, pages 370–379. IEEE Computer Society, 1998. 20

[41] J. Ziv and A. Lempel. **A Universal Algorithm for Sequential Data Compression**. *IEEE Trans. Inf. Theor.*, **23**(3):337–343, September 2006. 20

[42] S. Héman. **Super-Scalar Database Compression between RAM and CPU Cache**. 2005. 20, 55, 98

[43] R. N. Williams. **An extremely fast Ziv-Lempel data compression algorithm**. In *[1991] Proceedings. Data Compression Conference*, pages 362–371, 1991. 22

[44] Renaud Delbru, Stephane Campinas, Krystian Samp, and Giovanni Tummarello. **ADAPTIVE FRAME OF REFERENCE FOR COMPRESSING INVERTED LISTS**. 08 2020. 22

[45] Renaud Delbru, Stephane Campinas, and Giovanni Tummarello. **Searching Web Data: an Entity Retrieval and High-Performance Indexing Model**. *Journal of Web Semantics*, **10**, 01 2012. 22

[46] Robert Giegerich. **A discipline of dynamic programming over sequence data**. *Science of Computer Programming*, **51**:215–263, 06 2004. 22

[47] Vo Anh and Alistair Moffat. **Index compression using 64-bit words**. *Softw., Pract. Exper.*, **40**:131–147, 02 2010. 23, 25

[48] Daniel Lemire, Leonid Boytsov, and Nathan Kurz. **SIMD Compression and the Intersection of Sorted Integers**. *Software: Practice and Experience*, **46**, 01 2014. 23, 25, 26

[49] Fabrizio Silvestri and Rossano Venturini. **VSEncoding: Efficient Coding and Fast Decoding of Integer Lists via Dynamic Programming**. pages 1219–1228, 01 2010. 23

[50] Hao Yan, Shuai Ding, and Torsten Suel. **Inverted index compression and query processing with optimized document ordering**. pages 401–410, 01 2009. 23

[51] Naiyong Ao, Fan Zhang, Di Wu, Douglas Stones, Gang Wang, Xiaoguang Liu, Jing Liu, and Sheng Lin. **Efficient Parallel Lists Intersection and Index Compression Algorithms using Graphics Processing Units.** *PVLDB*, **4**:470–481, 05 2011. 24

[52] Wee Keong Ng and Chinya V. Ravishankar. **Block-Oriented Compression Techniques for Large Statistical Databases**. *IEEE Trans. on Knowl. and Data Eng.*, **9**(2):314–328, March 1997. 25

[53] Wangda Zhang, Yanbin Wang, and Kenneth Ross. **Parallel Prefix Sum with SIMD**. 09 2020. 27, 59

[54] Abdullah Al Hasib, Juan Cebrian, and Lasse Natvig. **V-PFORDelta: Data Compression for Energy Efficient Computation of Time Series**. pages 416–425, 12 2015. 27

[55] Patrick Damme, Dirk Habich, Juliana Hildebrandt, and Wolfgang Lehner. **Lightweight Data Compression Algorithms: An Experimental Survey (Experiments and Analyses)**. In *EDBT*, 2017. 28

[56] Peter Boncz, Viktor Leis, and Thomas Neumann. **ast Static Symbol Table (FSST): fast text compression that allows random access**. 08 2020. 29

[57] Wayne Zhao, Xudong Zhang, Daniel Lemire, Dongdong Shan, Jian-yun Nie, Hongfei Yan, and Ji-Rong Wen. **A General SIMD-Based Approach to Accelerating Compression Algorithms**. *ACM Transactions on Information Systems*, **33**, 02 2015. 29

[58] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. **Dremel: Interactive Analysis of Web-Scale Datasets**. *Commun. ACM*, **54**:114–123, 06 2011. 31

[59] Aditya Agarwal, Mark Slee, and Marc Kwiatkowski. **Thrift: Scalable Cross-Language Services Implementation**. Technical report, Facebook, 4 2007. 31

[60] Yongqiang He, Rubao Lee, Yin Huai, Zheng Shao, Namit Jain, Xiaodong (Frank) Zhang, and Zhiwei Xu. **RCFile: A fast and space-efficient data placement structure in MapReduce-based warehouse systems**. pages 1199–1208, 04 2011. 33

[61] Biswapesh Chattopadhyay, Priyam Dutta, Weiran Liu, Ott Tinn, Andrew McCormick, Aniket Mokashi, Paul Harvey, Hector Gonzalez, David Lomax, Sagar Mittal, Roee Aharon Ebenstein, Nikita Mikhaylin, Hung ching Lee, Xiaoyan Zhao, Guanzhong Xu, Luis Antonio Perez, Farhad Shahmohammadi, Tran Bui, Neil McKay, Vera Lychagina, and Brett Elliott. **Procella: Unifying serving and analytical data at YouTube**. *PVLDB*, **12(12)**:2022–2034, 2019. 35

[62] Animesh Trivedi, Patrick Stuedi, Jonas Pfefferle, Adrian Schuepbach, and Bernard Metzler. **Albis: High-performance file format for big data systems**. In *Proceedings of the 2018 USENIX Annual Technical Conference, USENIX ATC 2018*, Proceedings of the 2018 USENIX Annual Technical Conference, USENIX ATC 2018, pages 615–629. USENIX Association, 1 2020. 36

[63] Ahmed Zekri. **Enhancing the Matrix Transpose Operation Using Intel Avx Instruction Set Extension**. *International Journal of Computer Science Information Technology*, **6**:67–78, 06 2014. 38

[64] Elena Limonova, Arseny Terekhin, Dmitry Nikolaev, and Vladimir Arlazarov. **Fast Implementation of Morphological Filtering Using ARM NEON Extension**. 02 2020. 38

Appendices

A.1 C-PFOR

#	Dataset	column	pfor_r	c-pfor_r	diff
1	Bimbo_1	6	21.38	20.99	0.02
2	Bimbo_1	8	2.00	2.04	1.99
3	CityMaxCapita_1	7	2.02	2.04	0.07
4	CityMaxCapita_1	9	2.16	2.39	1.55
5	CityMaxCapita_1	10	2.43	2.59	0.42
6	CityMaxCapita_1	26	1.70	1.80	12.57
7	CommonGovernment_1	21	2.61	3.01	3.89
8	CommonGovernment_1	22	2.61	3.01	3.25
9	CommonGovernment_1	28	2.37	3.90	1.66
10	CommonGovernment_1	46	1.79	2.38	-7.29
11	CommonGovernment_10	21	2.61	2.97	3.56
12	CommonGovernment_10	22	2.61	2.97	3.61
13	CommonGovernment_10	28	2.37	3.83	1.57
14	CommonGovernment_10	46	1.79	2.34	-6.45
15	CommonGovernment_11	21	2.62	2.98	5.88
16	CommonGovernment_11	22	2.62	2.98	3.96
17	CommonGovernment_11	28	2.37	3.86	1.53
18	CommonGovernment_11	46	1.79	2.38	-7.66
19	CommonGovernment_12	21	2.62	2.97	3.67
20	CommonGovernment_12	22	2.62	2.97	4.13
21	CommonGovernment_12	28	2.37	3.79	1.62
22	CommonGovernment_12	46	1.79	2.36	-12.78
23	CommonGovernment_13	21	2.62	2.98	3.38
24	CommonGovernment_13	22	2.62	2.98	3.56
25	CommonGovernment_13	28	2.37	3.98	4.58
26	CommonGovernment_13	46	1.79	2.36	-5.39
27	CommonGovernment_2	21	2.62	2.97	3.79
28	CommonGovernment_2	22	2.62	2.97	3.64
29	CommonGovernment_2	28	2.37	3.85	1.32
30	CommonGovernment_2	46	1.79	2.36	-7.48
31	CommonGovernment_3	21	2.62	2.98	3.20
32	CommonGovernment_3	22	2.62	2.98	2.29

#	Dataset	column	pfor_r	c-pfor_r	diff
33	CommonGovernment_3	28	2.37	3.85	1.07
34	CommonGovernment_3	46	1.79	2.36	-9.80
35	CommonGovernment_4	21	2.62	2.96	5.28
36	CommonGovernment_4	22	2.62	2.96	5.28
37	CommonGovernment_4	28	2.37	3.83	0.43
38	CommonGovernment_4	46	1.79	2.33	-7.32
39	CommonGovernment_5	21	2.62	2.97	4.06
40	CommonGovernment_5	22	2.62	2.97	2.99
41	CommonGovernment_5	28	2.37	3.76	2.69
42	CommonGovernment_5	46	1.79	2.35	-7.37
43	CommonGovernment_6	21	2.63	2.99	4.14
44	CommonGovernment_6	22	2.63	2.99	4.06
45	CommonGovernment_6	28	2.37	3.85	4.00
46	CommonGovernment_6	46	1.79	2.35	-7.00
47	CommonGovernment_7	21	2.62	2.99	3.02
48	CommonGovernment_7	22	2.62	2.99	4.23
49	CommonGovernment_7	28	2.37	3.81	1.89
50	CommonGovernment_7	46	1.79	2.33	-9.01
51	CommonGovernment_8	21	2.62	3.01	3.44
52	CommonGovernment_8	22	2.62	3.01	3.54
53	CommonGovernment_8	28	2.37	3.81	4.71
54	CommonGovernment_8	46	1.79	2.35	-8.28
55	CommonGovernment_9	21	2.61	3.00	3.71
56	CommonGovernment_9	22	2.61	3.00	1.64
57	CommonGovernment_9	28	2.37	3.78	10.36
58	CommonGovernment_9	46	1.79	2.35	-7.28
59	Corporations_1	20	4.35	6.98	1.15
60	Corporations_1	25	24.29	23.59	-0.03
61	Eixo_1	9	2.99	3.44	2.19
62	Eixo_1	19	3.50	3.60	2.65
63	Food_1	2	1.87	1.92	4.14
64	Generico_1	7	1.77	1.85	3.46
65	Generico_2	7	1.77	1.84	-1.32
66	Generico_3	7	1.77	1.84	-0.90
67	Generico_4	7	1.77	1.85	-1.52
68	Generico_5	7	1.77	1.85	-1.14
69	HashTags_1	50	2.16	2.26	1.90
70	HashTags_1	51	2.27	2.43	1.41
71	HashTags_1	52	2.53	2.67	0.55
72	HashTags_1	57	3.80	4.37	-0.32
73	HashTags_1	63	1.76	1.85	8.63
74	Hatred_1	8	1.92	2.00	5.86
75	Hatred_1	9	2.26	2.32	0.75
76	Hatred_1	10	2.42	2.52	-0.42
77	Hatred_1	29	1.74	1.76	-3.47

#	Dataset	column	pfor_r	c-pfor_r	diff
78	IGlocations1_1	9	1.76	1.79	-3.63
79	IGlocations2_1	8	3.64	3.78	2.46
80	IGlocations2_2	8	3.64	3.78	1.63
81	IUBLibrary_1	4	3.60	3.75	1.74
82	IUBLibrary_1	16	3.59	3.75	1.71
83	Medicare1_1	21	2.04	2.06	0.47
84	Medicare1_2	21	2.04	2.06	0.32
85	Motos_1	7	1.77	1.84	-0.61
86	Motos_2	7	1.77	1.85	0.42
87	MulheresMil_1	9	2.99	3.43	2.28
88	MulheresMil_1	19	3.49	3.58	2.50
89	RealEstate1_1	16	1.67	1.73	0.82
90	RealEstate1_2	16	1.67	1.74	0.14
91	RealEstate2_1	17	1.61	1.60	-4.92
92	RealEstate2_2	17	1.61	1.59	-5.50
93	RealEstate2_3	17	1.61	1.59	-5.30
94	RealEstate2_4	17	1.61	1.60	-4.99
95	RealEstate2_5	17	1.61	1.59	-2.55
96	RealEstate2_6	17	1.61	1.60	-5.44
97	RealEstate2_7	17	1.61	1.60	-5.07
98	Rentabilidad_1	62	3.32	4.87	7.18
99	Rentabilidad_1	63	1.57	1.90	4.28
100	Rentabilidad_1	131	3.68	5.67	7.16
101	Rentabilidad_1	139	3.71	5.74	7.32
102	Rentabilidad_2	60	3.32	4.86	7.26
103	Rentabilidad_2	61	1.57	1.89	4.09
104	Rentabilidad_2	129	3.68	5.66	7.24
105	Rentabilidad_2	137	3.71	5.73	7.21
106	Rentabilidad_3	62	3.32	4.86	7.46
107	Rentabilidad_3	63	1.57	1.89	4.49
108	Rentabilidad_3	131	3.68	5.66	6.81
109	Rentabilidad_3	139	3.71	5.73	7.04
110	Rentabilidad_4	62	3.32	4.86	7.37
111	Rentabilidad_4	63	1.57	1.88	3.70
112	Rentabilidad_4	131	3.68	5.66	6.82
113	Rentabilidad_4	139	3.71	5.73	7.03
114	Rentabilidad_5	62	3.32	4.85	7.19
115	Rentabilidad_5	63	1.57	1.89	4.28
116	Rentabilidad_5	131	3.68	5.66	7.13
117	Rentabilidad_5	139	3.71	5.73	13.60
118	Rentabilidad_6	62	3.32	4.86	8.41
119	Rentabilidad_6	63	1.57	1.90	4.04
120	Rentabilidad_6	131	3.68	5.66	7.24
121	Rentabilidad_6	139	3.71	5.73	8.23
122	Rentabilidad_7	62	3.32	4.86	7.16

#	Dataset	column	pfor_r	c-pfor_r	diff
123	Rentabilidad_7	63	1.57	1.89	3.01
124	Rentabilidad_7	131	3.68	5.66	6.37
125	Rentabilidad_7	139	3.71	5.73	0.36
126	Rentabilidad_8	62	3.32	4.86	7.17
127	Rentabilidad_8	63	1.57	1.89	5.57
128	Rentabilidad_8	131	3.68	5.66	7.16
129	Rentabilidad_8	139	3.71	5.73	7.02
130	Rentabilidad_9	62	3.32	4.86	7.13
131	Rentabilidad_9	63	1.57	1.90	3.27
132	Rentabilidad_9	131	3.68	5.66	7.37
133	Rentabilidad_9	139	3.71	5.73	7.07
134	TableroSistemaPenal_2	5	1.60	1.73	7.72
135	USCensus_1	277	2.29	2.28	-0.10
136	USCensus_1	278	2.29	2.28	0.01
137	USCensus_1	315	1.52	1.52	0.54
138	USCensus_1	316	1.52	1.52	0.50
139	USCensus_2	277	2.29	2.28	-1.45
140	USCensus_2	278	2.29	2.28	0.01
141	USCensus_2	315	1.52	1.52	-0.21
142	USCensus_2	316	1.52	1.52	-0.44
143	USCensus_3	277	2.29	2.28	0.08
144	USCensus_3	278	2.29	2.28	-0.04
145	USCensus_3	315	1.52	1.52	0.17
146	USCensus_3	316	1.52	1.52	0.87
147	Uberlandia_1	9	2.99	3.44	2.00
148	Uberlandia_1	15	2.13	2.32	1.19
149	Uberlandia_1	19	3.50	3.58	2.41
150	Wins_1	40	1.91	1.90	0.33
151	Wins_1	41	1.94	1.93	0.13
152	Wins_2	154	1.91	1.90	0.45
153	Wins_2	155	1.94	1.93	0.17
154	Wins_2	484	12.88	13.37	-0.03
155	Wins_3	154	1.91	1.90	0.85
156	Wins_3	155	1.94	1.93	0.78
157	Wins_3	484	12.88	13.37	0.09
158	Wins_4	154	1.91	1.90	0.59
159	Wins_4	155	1.94	1.93	0.37
160	Wins_4	484	12.88	13.37	-0.16

Table 1: Difference between scenario A and B in terms of compression ratio.

A.2 Literature Study

	Rel.	TITLE	I1	I2	I3	I4	I5	i6	I7	I8	E1	Included?
1	"++"	Super-Scalar RAM-CPU Cache Compression				x						true
2	"++"	Super-Scalar Database Compression between RAM and CPU Cache				x						true
3	"++"	Fast integer compression using SIMD instructions				x						true
4	"++"	SIMD-Scan: Ultra Fast in-Memory Table Scan using onChip Vector Processing Units			x							true
5	"++"	BitWeaving: Fast scans for main memory data processing							x			true
6	"+"	Efficient Lightweight Compression Alongside Fast Scans				x						true
7	"++"	Compressing Relations and Indexes				x						true
8	"+"	Searching Web Data: an Entity Retrieval and High-Performance Indexing Model				x						true
9	"++"	Inverted index compression and query processing with optimized document ordering.				x						true
10	"+	Efficient Parallel Lists Intersection and Index Compression Algorithms using Graphics Processing Units.				x						true
11	"++"	Decoding billions of integers per second through vectorization				x						true
12	"++"	SIMD Compression and the Intersection of Sorted Integers				x						true

13	"++"	Block-Oriented Compression Techniques for Large Statistical Databases		x				true
14	"++"	V-PFORDelta: Data Compression for Energy Efficient Computation of Time Series		x				true
15	"+"	Lightweight Data Compression Algorithms: An Experimental Survey (Experiments and Analyses).		x				true
16	"+"	How to speed Connected Component Labeling up with SIMD RLE algorithms		x				true
17	"+"	A Decomposition Storage Model				x		true
18	"++"	Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation	x					true
19	"++"	Parallel Prefix Sum with SIMD		x				true
20	"-"	Fast Implementation of Morphological Filtering Using ARM NEON Extension					x	true
21	"++"	Dremel: Interactive Analysis of Web-Scale Datasets			x			true
22	"++"	Enhancing the Matrix Transpose Operation Using Intel Avx Instruction Set Extension					x	true
23	"++"	Integrating Compression and Execution in Column-Oriented Database Systems		x				true
24	"++"	White-box Compression: Learning and Exploiting Compact Table Representations			x			true

25	"++"	Albis: High-Performance File Format for Big Data Systems				x			true
26	"++"	Weaving Relations for Cache Performance		x					true
27	"++"	Procella: Unifying serving and analytical data at YouTube				x			true
28	"++"	Monkey: Optimal Navigable Key-Value Store						x	false
29	"++"	WideTable: An Accelerator for Analytical Data Processing						x	false
30	"++"	Vectorizing Database Column Scans with Complex Predicates			x				true
31	"++"	Major Technical Advancements in Apache Hive				x			true
32	"++"	RCFile: A fast and space-efficient data placement structure in MapReduce-based warehouse systems						x	false
33	"+"	High-Throughput Big Data Analytics Through Accelerated Parquet to Arrow Conversion	x						true
34	"++"	Nozzle: A Schemaon-Read Parquet						x	false
35	"++"	Characterization of a Big Data Storage Workload in the Cloud	x						true
36	"++"	Self-learning Whitebox Compression				x			true
37	"+"	Beyond Straightforward Vectorization of Lightweight Data Compression Algorithms for Larger Vector Sizes			x				true

38	"+"	Conflict Detection-Based Run-Length Encoding - AVX-512 CD Instruction Set in Action	x			true
39	"+"	PIDS: Attribute Decomposition for Improved Compression and Query Performance in Columnar Storage		x		true
40	"++"	FSST: Fast Random Access String Compression	x			true
41	"-"	DSM vs. NSM: CPU performance tradeoffs in block-oriented query processing			x	true
42	"++"	Index compression using 64-bit words	x			true
43	"++"	VSEncoding: efficient coding and fast decoding of integer lists via dynamic programming.	x			true
44	"++"	A General SIMD-based Approach to Accelerating Compression Algorithms	x			true
45	"++"	ADAPTIVE FRAME OF REFERENCE FOR COMPRESSING INVERTED LISTS	x			true

Table 2: Literature inclusion table.

	Inclusion and Exclusion Criterion
I1	A study that reviews or analyzes big data file formats
I2	A study that proposes a new storage layout for columnar storage
I3	A study that proposes an improvment to bit-(un)packing
I4	A study that proposes an improvment to light-weight compression algorithmes such as FOR, differential coding, PFOR, PDELTA, PDICT, PFORDELTA and RLE
I5	A study that uses, extends or is related to the concept of Whitebox compression
I6	A study that designs a new big data file format
I7	A study that proposes or analyzes a storage layout for databas systems
I8	A study that SIMDize the matrix transpose operator
E1	A study which is related to other aspects of big data file formats (not related to storage and compression)

Table 3: Inclusion and exclusion criteria